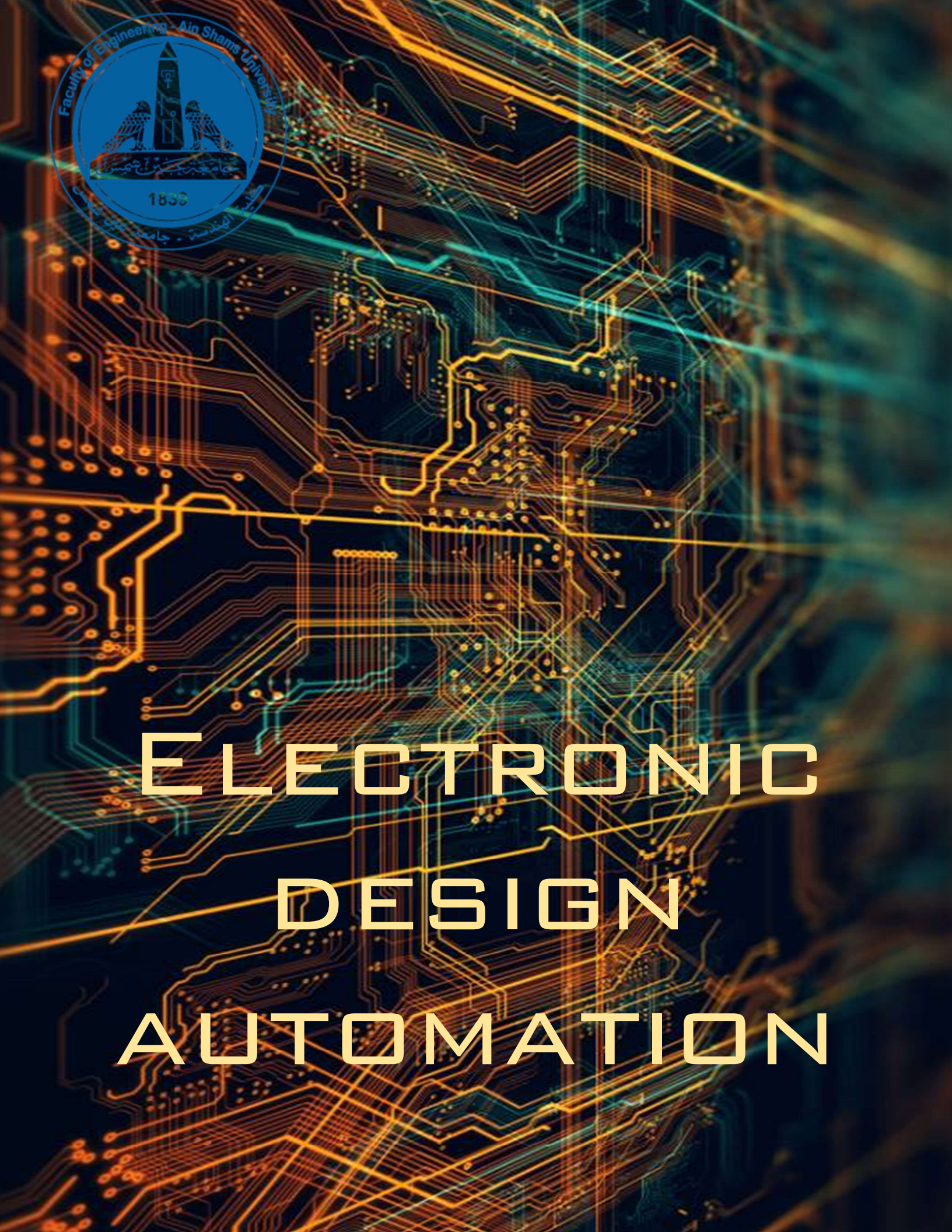

Faculty of Engineering - Ain Shams University
1839
Electronic
Design
Automation

AIN SHAMS UNIVERSITY FACULTY OF ENGINEERING 1-CREDIT HOURS ENGINEERING PROGRAMS COMPUTER ENGINEERING AND SOFTWARE SYSTEMS PROGRAM

2023-2024

Verilog Lint Project

Electronic Design Automation

CSE 312

Name	**ID**
Ahmed Nezar Ahmed Hussien	21P0025
AbdulRahman Hesham Kamel Seleim Abduallah	21P0153
Kirollos Ehab Magdy Halim	21P0006
Farida Waleed Mohamed Mohamed Fakhry	21P0167
Yasmina Nasser Hamam Abdelfadeel	21P0211
Monica Hany Makram Derias	21P0173
Patrick Ramez Eskander Bolous	21P0180

Course Coordinator:

Dr. Eman El Mandoh & Eng. Kareem Waseem

Contents

1 Abstract

The project aims to address the imperative need for enhanced reliability and correctness in digital hardware designs by introducing a powerful Static Verilog Design Checker. Focused on the Design Under Test (DUT) and unburdened by the necessity of a testbench, VeriGuard aims to identify and pinpoint potential design flaws early in the development cycle. This tool's primary objectives include detecting critical violations such as Arithmetic Overflow, Unreachable Blocks, Unreachable FSM State, Un-initialized Registers, Multi-Driven Bus/Registers, Non Full/Parallel Case, and the elusive Infer Latch. Operating on a single Flatten Module, VeriGuard embraces simplicity, ensuring an efficient approach to static analysis in the area of Verilog-based hardware design.

2 Introduction

The VeriGuard project aims for reliability and correctness of Verilog code and The ensure the early detection of potential design flaws.

2.1 Project Structure

The VeriGuard project is composed of:

1. **Parser:** The parser component is responsible for analyzing Verilog code, extracting relevant information, and preparing it for further analysis. It serves as the initial step in the static analysis process.
2. **Static Checker Engine:** The core of VeriGuard, the static checker engine, houses the algorithms and logic necessary for detecting various violations within the parsed Verilog code.
3. **Report Generator:** The report generator is a crucial component that transforms the results of the static analysis into a comprehensible format. It produces a text file demonstrating essential information, including the checks' names, the specific lines of code where violations occur, and the signals/variables impacted.

2.2 Project Objectives

The core objectives of VeriGuard are to detect and highlight various critical violations that could compromise the integrity of a digital hardware design. These violations encompass the following:

1. **Arithmetic Overflow:** This violation occurs when arithmetic operations result in a value that exceeds the maximum representable size for a given data type.
2. **Unreachable Blocks:** Identifying unreachable code blocks is crucial for eliminating redundant sections of the design that may never be executed.
3. **Unreachable FSM State:** In the context of Finite State Machines (FSMs), VeriGuard ensures that all defined states are reachable. Detection of unreachable FSM states is vital for preventing unintended and potentially hazardous system behavior.
4. **Un-initialized Registers:** Detecting un-initialized registers is crucial to ensuring predictable and stable system behavior from the outset.
5. **Multi-Driven Bus/Registers:** Addressing the complexities of bus and register interactions, Identifying instances where a bus or register is driven by multiple conflicting sources. This detection prevents contention issues and enhances design robustness.
6. **Non Full/Parallel Case:** Analyzing case statements to ensure completeness and parallelism. Detection of non-full or non-parallel cases ensures that all possible conditions are considered, avoiding unintended consequences due to incomplete case coverage.

3 Verilog Cases

In simpler terms, this property ensures that when the system is in the state of displaying the balance (state == 2) and the operation is a balance display (operation == 3), the next state should have a balance equal to the balance stored in the database for the account index preceding the current account index. This is a common kind of property used to check correctness in digital design verification.

3.1 Edge Cases module

Code Snippet:

```
module Edge_Cases (data_out);
    output reg data_out;
    reg A;

    always @(*)
    begin
        case (A):
            1'b0: A = 1'b1;
            1'b1: A = 1'b0;
        endcase
    end
endmodule
```

Description:

- **Module Declaration:** The module is named `Edge_Cases`, with a single output port `data_out`.
- **Port Definition:**
  - `data_out` is an output port declared as a register (`reg`), used for storing the output value.
- **Internal Register:**
  - `A` is a register used in the module.
- **Always Block:**
  - Triggered on any change in the sensitivity list (`*`).
  - Contains a case statement that updates the value of `A` based on its current value.
- **Output Logic:**
  - The module doesn't have a direct output (`data_out`). It seems the intention is to modify the value of `A` based on the case statement.

3.2 Vector Input module

Code Snippet:

```
module Vector_Input (data_out);
    output reg [1:0] data_out;
    reg [1:0] A;

    always @(*)
    begin
        case (A):
            2'b00: A = 2'b1;
            2'b01: A = 2'b0;
        endcase
    end
endmodule
```

Description:

- **Module Declaration:** The module is named `Vector_Input`, with a single output port `data_out`.
- **Port Definition:**
  - `data_out` is an output port declared as a vector (`reg [1:0]`), used for storing the output value.
- **Internal Register:**
  - `A` is a vector register used in the module (`reg [1:0]`).
- **Always Block:**
  - Triggered on any change in the sensitivity list (`*`).
  - Contains a case statement that updates the value of `A` based on its current value.
- **Output Logic:**
  - The module updates the value of `A` based on the case statement, effectively swapping the bits in the vector.

3.3 CaseZ Parallel Case Module

Code Snippet:

```
module CaseZ_Parallel_Case(A);
    input reg [3:0] A;
    reg [1:0] F;

    always @(*)
    begin
        casez(A)
            4'b???1: F = 2'b00;
            4'b??1?: F = 2'b01;
            4'b?1??: F = 2'b10;
            4'bl???: F = 2'b11;
        endcase
    end
endmodule
```

Description:

- **Module Declaration:** The module is named `CaseZ_Parallel_Case` with a single input port `A` and an output register `F` (2 bits).
- **Port Definition:**
  - `A`: Input port, a 4-bit register.
  - `F`: Output register, 2 bits.
- **Always Block:**
  - Triggered on any change in `A`.
  - Uses `casez` statement to define logic based on the patterns in `A`.

3.4 Sequential Case Module

Code Snippet:

```
module Sequential_Case(t);
    input reg [3:0] t;

    always @(*)
    begin
        case (t)
            4'b0000: t = 4'b0001;
            4'b0001: t = 4'b0010;
        endcase
    end
endmodule
```

Description:

- **Module Declaration:** The module is named `Sequential_Case` with a single input port `t`.
- **Port Definition:**
  - `t`: Input port, a 4-bit register.
- **Always Block:**
  - Triggered on any change in `t`.
  - Uses a `case` statement to define sequential logic based on the value of `t`.
  - When `t` is `4'b0000`, it updates `t` to `4'b0001`.
  - When `t` is `4'b0001`, it updates `t` to `4'b0010`.
  - There is no default case or break statement, so this logic will continuously execute for any other values of `t`, potentially leading to an infinite loop.

3.5 Unreachable Blocks module

Code Snippet:

```
module UnreachableBlocks(data_out);
    output reg data_out;
    reg reach;
    wire state;

    initial
    begin
        reach = 1'b1;
    end

    always @(state)
    begin
        if (reach == 2'b0)
        begin
            data_out = 1'b1;
        end
        else
        begin
            data_out = 1'b0;
        end
    end
endmodule
```

Description:

- **Module Declaration:** The module is named `UnreachableBlocks`, with a single output port `data_out` declared as a register.
- **Port Definition:**
  - `data_out` is an output register used for storing the output value.
- **Internal Register:**
  - `reach` is a register initialized to `1'b1` and controls the output.
- **Initial Block:**
  - Sets the initial value of `reach` to `1'b1`.
- **Always Block:**
  - Triggered on changes of `state`. Contains a conditional statement checking `reach`. The condition `reach == 2'b0` is unreachable due to a bit size mismatch.
- **Output Logic:**
  - `data_out` is set to `1'b1` if the unreachable condition is met; otherwise, it's set to `1'b0`.

3.6 Uninitialized Register module

Code Snippet:

```
module UninitializedRegister(data_out);
    reg data;
    output reg data_out;
    assign data_out = data;
endmodule
```

Description:

- **Module Declaration**:
  - The module is named `UninitializedRegister`, primarily used for simulation or synthesis in digital design.
- **Register and Output Declaration**:
  - `data`: A register (`reg`) that serves as an internal storage element. It is not initialized within the module, meaning its initial value is indeterminate.
  - `data_out`: An output port declared as a register (`reg`). It is used to store and output the value of `data`.
- **Continuous Assignment**:
  - The statement `assign data_out = data;` is a continuous assignment that directly connects the value of the `data` register to the `data_out` output. This means `data_out` will always reflect the current value of `data`.
- **Lack of Initialization**:
  - The `data` register is not initialized within the module. This can lead to unpredictable behavior in simulation and synthesis because its initial value is unknown.
- **No Input or Control Logic**:
  - The module lacks input ports or any internal logic to modify or control the value of `data`. Without external intervention or additional code, `data` will remain at its uninitialized value.
- **Potential Synthesis Issues**:
  - In a synthesis context, uninitialized registers can be a source of design issues. Synthesis tools might treat such registers in different ways, potentially leading to inconsistent behavior.
- **Usage Context**:
  - This module could be part of a larger design or used for educational purposes to demonstrate the behavior of uninitialized registers. However, in practical applications, it's important to initialize registers to ensure predictable and reliable behavior.

3.7 Inferring Latches module

Code Snippet:

```
module InferringLatches(enable, Data, out);
    input wire enable, Data
    output reg out;

    always @(enable)
    begin
        if (enable)
        begin
            out = Data;
        end
    end
endmodule
```

Description:

- **Module Declaration**:
  - The module is called `InferringLatches`, suitable for use in digital design simulations or synthesis.
- **Port Definitions**:
  - `enable`: An input port of type `wire`, acting as a control signal.
  - `Data`: Another input port of type `wire`, representing the data input.
  - `out`: An output port declared as a register (`reg`), used for storing and outputting the data value.
- **Latch Inference**:
  - The `always @(enable)` block triggers only on changes to the `enable` signal. This is indicative of a level-sensitive latch.
  - Within this block, `out` is assigned the value of `Data` only when `enable` is high (`if (enable)`). When `enable` is low, `out` retains its previous value, which is characteristic behavior of a latch.
- **Partial Sensitivity List**:
  - The sensitivity list of the `always` block includes only `enable` and not `Data`. This means changes in `Data` will not trigger the block unless `enable` also changes.
- **Implicit Latch Creation**:
  - Due to the partial sensitivity list and the conditional assignment inside the `always` block, this code infers the creation of a latch, rather than a flip-flop. This is an important distinction in digital design, as latches are level-sensitive and can lead to different behavior than edge-triggered flip-flops.
- **Design Considerations**:
  - Latches are often inferred unintentionally in Verilog and can lead to design issues like increased power consumption, timing problems, and difficulties in meeting timing constraints in synchronous designs.
  - It's generally recommended to avoid unintentional latch creation in synchronous designs and to be explicit about latch usage.

3.8 Unreachable State module

Code Snippet:

```
module UnreachableState(clk, state_out);
    input clk;
    output reg [1:0] state_out;
    reg [1:0] current_state, next_state;
    localparam [1:0] S1 = 2'b00 ;
    localparam [1:0] S2 = 2'b01 ;
    localparam [1:0] S3 = 2'b10 ;

    always @(posedge clk)
    begin
        current_state <= next_state;
    end

    always @(*)
    begin
        case (current_state)
            S1:
            begin
            next_state <= S2;
            end
            S2:
            begin
                next_state <= S1;
            end
            S3:
            begin
                next_state <= S1;
            end
        endcase
        state_out = current_state;

    end
endmodule
```

Description:

- **Module Declaration**:
  - Named `UnreachableState`, with `clk` as an input and `state_out` as a 2-bit output.
- **Input and Output**:
  - `clk`: An input clock signal.
  - `state_out`: A 2-bit output register that holds the current state of the state machine.
- **Registers**:
  - `current_state` and `next_state`: 2-bit registers used to hold the current and next states of the state machine.

- **State Definitions**:
  - `S1`, `S2`, and `S3` are local parameters representing three states of the state machine, encoded as 2-bit values.
- **Sequential Logic Block**:
  - An `always @(posedge clk)` block updates `current_state` with `next_state` at every positive edge of the clock. This block is responsible for the state transition.
- **Combinational Logic Block**:
  - An `always @(*)` block determines the next state based on the current state. It uses a `case` statement to define state transitions:
    - From `S1` to `S2`.
    - From `S2` to `S1`.
    - From `S3` to `S1`.
  - `state_out` is set to the value of `current_state`, reflecting the current state externally.
- **Issue with Unreachable State**:
  - The state `S3` is never reached in normal operation. The state machine transitions only between `S1` and `S2`. As there are no conditions or paths leading to `S3`, it remains an unreachable state in this design.
- **Lack of Initialization**:
  - The registers `current_state` and `next_state` are not initialized. Without initialization, the starting state of the state machine is indeterminate, which could lead to unpredictable behavior.
- **Design Implications**:
  - The unreachable state (`S3`) indicates either an oversight in the state transition logic or an incomplete design. It's essential to ensure that all defined states are reachable and serve a purpose in the state machine.
  - For practical applications, it's important to initialize the state machine to a known state, often the initial or reset state.

3.9 Incomplete Case Module

Code Snippet:

```
module Incomplete_Case(y_out);
    output reg [1:0] y_out;
    reg [1:0] x, y;

    always @(*)
    begin
        case(x)
            2'b00: y = 1'b00;
            2'b01: y = 1'b01;
        endcase
        y_out = y;
    end
endmodule
```

Description:

- **Module Declaration:** The module is named `Incomplete_Case` with a single output port `y_out`.
- **Port Definition:**
  - `y_out`: Output port declared as a 2-bit register (`reg`).
- **Internal Registers:**
  - `x` and `y` are 2-bit registers.
- **Always Block:**
  - Triggered on any change in `x`.
  - Uses a `case` statement to define logic based on the value of `x`.
  - Defines cases for `2'b00` and `2'b01`.
  - Missing cases for `2'b10` and `2'b11` are commented as incomplete.
- **Output Logic:**
  - Assigns the value of `y` to `y_out`.

3.10 Non-Parallel Z Module

Code Snippet:

```
module NonParallelZ(x);
    input [1:0] x;
    reg [1:0] y;
    always @(*)
    begin
        casez (x)
            2'b??: y = 1'b00;
            2'b??: y = 1'b01;
            2'b??: y = 1'b10;
            2'b??: y = 1'b11;
        endcase
    end
endmodule
```

Description:

- **Module Declaration:** The module is named `NonParallelZ` with a single input port `x`.
- **Port Definition:**
  - `x`: Input port, a 2-bit vector.
- **Internal Registers:**
  - `y` is a 2-bit register.
- **Always Block:**
  - Triggered on any change in `x`.
  - Uses a `casez` statement to define logic based on the pattern in `x`.
  - Cases are defined with wildcard (`?`) for non-specified bits in `x`.
  - Each case assigns a specific value to `y`.

3.11 Non-Parallel X Module

Code Snippet:

```
module NonParallelX(x);
    input [1:0] x;
    reg [1:0] y;
    always @(*)
    begin
        casex (x)
            2'bxx: y = 1'b00;
            2'bxx: y = 1'b01;
            2'bxx: y = 1'b10;
            2'bxx: y = 1'b11;
        endcase
    end
endmodule
```

Description:

- **Module Declaration:** The module is named `NonParallelX` with a single input port `x`.
- **Port Definition:**
  - `x`: Input port, a 2-bit vector.
- **Internal Registers:**
  - `y` is a 2-bit register.
- **Always Block:**
  - Triggered on any change in `x`.
  - Uses a `casex` statement to define logic based on the pattern in `x`.
  - Cases are defined with wildcard (`x`) for non-specified bits in `x`.
  - Each case assigns a specific value to `y`.

3.12 Full Case Module

Code Snippet:

```
module Full_Case(y_out);
    output reg [1:0] y_out;
    reg [1:0] x, y;

    always @(*)
    begin
        case(x)
            2'b00: y = 1'b00;
            2'b0?: y = 1'b01;
            2'b?0: y = 1'b10;
            default: y = 1'b11;
        endcase
        y_out = y;
    end
endmodule
```

Description:

- **Module Declaration**:
  - The module is called `Full_Case`, designed for use in digital logic systems.
- **Output Declaration**:
  - `y_out`: A 2-bit output register used to output the value of `y`.
- **Internal Registers**:
  - `x` and `y`: 2-bit registers used for internal logic operations. Note that `x` is not assigned any input and is not initialized, which might lead to undefined behavior.
- **Combinational Logic Block**:
  - An `always @(*)` block, indicating combinational logic that updates its output whenever its input changes.
  - The `case` statement inside the block handles different values of `x`:
    - When `x` is `2'b00`, `y` is set to `1'b00`.
    - When the first bit of `x` is `0`, `y` is set to `1'b01`.
    - When the second bit of `x` is `0`, `y` is set to `1'b10`.
    - The `default` case is added to handle all other cases; `y` is set to `1'b11`.
  - `y_out` is assigned the value of `y` outside the case statement.

3.13 MultipleDrivers Module

Code Snippet:

```
module MultipleDrivers(myIn, outputVar);
    input [1:0] myIn;
    output reg [1:0] outputVar;
    reg myReg;

    always @(*)
    begin
        myReg = myReg + 1;
        myReg = 1'b0;
    end

    always @(*)
    begin
        outputVar = myIn;
    end
endmodule
```

Description:

- **Module Declaration:** The module is named `MultipleDrivers`, suitable for digital logic applications.
- **Input and Output:**
  - `myIn`: A 2-bit input vector.
  - `outputVar`: A 2-bit output register.
- **Internal Register:**
  - `myReg`: A 1-bit register used in the first always block.
- **First Always Block (Combinational Logic):**
  - Triggered by any changes in its inputs, as indicated by `@(*)`.
  - Contains conflicting assignments to `myReg`:
    - `myReg = myReg + 1;`: This line attempts to increment `myReg`, a sequential operation placed in a combinational block.
    - `myReg = 1'b0;`: This immediately resets `myReg` to 0 in the same always block, potentially overriding the previous increment operation.
  - The conflicting assignments can lead to race conditions or unpredictable behavior and are not recommended in combinational logic.
- **Second Always Block (Combinational Logic):**
  - Also triggered by any changes in its inputs.
  - Assigns the value of `myIn` to `outputVar`, a straightforward and correct operation for combinational logic.

3.14 MultipleDrivers2 Module Code

```
module MultipleDrivers2 (input [1:0] x, output out2);
    input [1:0] myIn;

    assign out2 = 0'b1;

    always @(*)
    begin
        out2 = myIn[0];
    end

endmodule
```

Description

- **Module Declaration:** The module is named `MultipleDrivers2`, suitable for digital logic applications.

- **Input and Output:**
  - `myIn`: A 2-bit input vector.
  - `outputVar`: A 2-bit output register.

- **Internal Register:**
  - `myReg`: A 1-bit register used in the first always block.

- **First Always Block (Combinational Logic):**
  - Triggered by any changes in its inputs, as indicated by `@(*)`.
  - Contains conflicting assignments to `myReg`:
    - `myReg = myReg + 1;`: This line attempts to increment `myReg`, a sequential operation placed in a combinational block.
    - `myReg = 1'b0;`: This immediately resets `myReg` to 0 in the same always block, potentially overriding the previous increment operation.
  - The conflicting assignments can lead to race conditions or unpredictable behavior and are not recommended in combinational logic.

- **Second Always Block (Combinational Logic):**
  - Also triggered by any changes in its inputs.
  - Assigns the value of `myIn` to `outputVar`, a straightforward and correct operation for combinational logic.

3.15 Arithmetic Overflow module

Code Snippet:

```
module ArithmeticOverflow(a,b,result);
    input reg [3:0] a, b;
    output reg [3:0] result;

    assign result = a + b;
endmodule
```

Description:

- **Module Declaration**:
  - The module is named `ArithmeticOverflow`, indicating its purpose in demonstrating arithmetic operations, specifically addition.
- **Input and Output**:
  - `a` and `b`: 4-bit input registers intended to hold the operands for the addition.
  - `result`: A 4-bit output register intended to store the result of the addition.
- **Arithmetic Operation**:
  - The line `assign result = a + b;` performs the addition of `a` and `b`.
- **Critical Issue - Overflow**:
  - The result of adding two 4-bit numbers can be a 5-bit number in case of an overflow. For example, adding `1111` (15 in decimal) and `0001` (1 in decimal) results in `10000` (16 in decimal), which is a 5-bit number.
  - However, `result` is declared as a 4-bit register, meaning it can only store the lower 4 bits of the addition result. This leads to an overflow issue where the most significant bit (MSB) of the result is lost or truncated, resulting in incorrect or unintended values.
- **Incompatible Assignment**:
  - The `assign` statement is used incorrectly here. The `assign` keyword is meant for continuous assignments to wires, not registers. Since `result` is declared as a `reg`, the correct approach would be to use an `always` block for the assignment.
- **Design Considerations and Fixes**:
  - To correctly handle potential overflows, `result` should be declared with sufficient bits to accommodate the maximum possible sum (in this case, 5 bits).
  - To resolve the incompatible assignment issue, change `result` to a wire or use an `always` block for the assignment.
  - If overflow detection is required, additional logic needs to be implemented to detect when the sum exceeds the capacity of the 4-bit `result`.

3.16 CombinationalFeedbackLoop Module

Code Snippet:

```
module CombinationalFeedbackLoop(a, b);
    input a;
    reg b;

    always @(*)
    begin
        b = b + a;
    end
endmodule
```

Description:

- **Module Declaration**:
  - The module is named `CombinationalFeedbackLoop`, indicating a combinational circuit with feedback.
- **Input and Output**:
  - `a`: Single-bit input.
  - `b`: Single-bit register, representing the output.
- **Combinational Logic Block**:
  - The `always @(*)` block signifies combinational logic, where the output is updated whenever the input (`a`) changes.
  - The logic operation `b = b + a;` adds the input `a` to the current value of `b` and assigns the result back to `b`.
  - This represents a feedback loop as the value of `b` is influenced by its own previous value.
- **Feedback Loop Considerations**:
  - Creating feedback loops in combinational logic is generally discouraged in digital design because it can lead to unpredictable behavior and timing issues.
  - In this case, changes in `b` immediately affect the calculation of the next value of `b`, potentially causing unexpected results.
- **Suggested Improvement**:
  - If sequential behavior or feedback is desired, consider using a clocked always block (`always @(posedge clk)` or similar) to ensure a well-defined and predictable update sequence.

4 Checkers

4.1 Verilog Parser in Python

The Verilog Parser, developed as a Python script, is designed to analyze and process Verilog code effectively. Its primary function is to parse the code, identifying and reporting common errors or deviations from established Verilog coding standards. By performing a detailed examination of the Verilog file, the script detects issues like unreachable blocks, uninitialized registers, inferring latches, and multiple drivers among others. This Python-based tool significantly contributes to ensuring code quality and adherence to the best practices prescribed for Verilog programming, thereby facilitating a more efficient and error-free code development process.

4.2 Components

4.2.1 Verilog Code Reader and Analyzer

The Verilog Code Reader and Analyzer is a critical component of the Python script, primarily responsible for opening and reading the Verilog file, and then systematically analyzing its contents. This component acts as the foundational step in parsing, breaking down the Verilog source code into manageable segments for further inspection and issue identification.

Functionality:

- **File Reading:** Opens and reads the Verilog file line by line, preparing the code for analysis.
- **Module Counting and Listing:** Counts the number of modules and organizes code into segments for individual analysis, facilitating the detailed inspection of each module's hierarchy and structure.
- **Comment Handling:** Strips out comments and cleans the code to ensure that only relevant Verilog constructs are analyzed, providing clarity and focus in subsequent steps.

Challenges Addressed:

- **Complex Syntax and Structure Handling:** Manages and interprets the various complex constructs of Verilog, such as module declarations, input/output ports, and signal assignments.
- **Robust Error Identification:** Effectively identifies common issues in the code, including unreachable blocks, uninitialized registers, and potential latches, among others. This helps in ensuring a thorough review against best practices and standard conventions in Verilog programming.

```python
def file_reader(file_name):
    x = []
    try:
        # Open the file in read mode
        with open(file_name, 'r') as file:
            # Read and print each line
            for line in file:
                x.append(line.strip())

    except FileNotFoundError:
        print(f"File not found: {file_name}")

    except Exception as e:
        print(f"An error occurred: {e}")

    return x


file_path = 'tst.v'

verilog_code = file_reader(file_path)

for i in verilog_code:
        print(i)
```

4.2.2 Static Checker Engine

The Static Checker Engine is the core analytical component responsible for identifying coding violations within the parsed Verilog code. It employs a rule-based approach, applying a set of predefined rules to catch common issues that may compromise the integrity and functionality of Verilog designs.

Functionality:

- **Issue Detection:** Employs a series of specialized functions to thoroughly scan the Verilog code for specific issues such as unreachable blocks, uninitialized registers, unintentional latches, and conflicting signal assignments among others.
- **Detailed Analysis:** Investigates both the structural and logical aspects of the Verilog code using pattern recognition and contextual analysis techniques to accurately pinpoint and flag coding irregularities.
- **Reporting Mechanism:** Consolidates all identified issues into a structured report, detailing the location, nature, and potential implications of each problem, thus providing a clear and actionable summary of findings.

Coding Violations Addressed:

1. **Unreachable Blocks:** Identifies code segments that are logically inaccessible due to the surrounding conditional or looping constructs, ensuring all parts of the code contribute to the design's functionality.
2. **Uninitialized Registers:** Flags any registers used within the code that lack a definite initial value, potentially leading to unpredictable design behavior.
3. **Multiple Drivers:** Highlights scenarios where a single signal is driven by multiple sources, creating conflicts and possibly erratic behavior in the resulting circuit.
4. **Inferring Latches:** Detects areas in the code that may unintentionally infer latches, particularly in scenarios involving incomplete conditional statements or missing cases in combinational blocks.
5. **Arithmetic Overflow:** Scans for potential overflow scenarios in arithmetic computations, a common source of subtle bugs in digital logic.
6. **Non-Full and Non-Parallel Cases:** Identifies case statements that either do not cover all possible scenarios or have overlapping conditions, leading to potential nondeterminism or inefficiencies.

Flexibility:

- **Custom Checks and Extensions:** Designed with adaptability in mind, the script allows for the integration of new checks or modification of existing ones to cater to the specific needs of different Verilog projects, ensuring the tool's effectiveness and relevance across a wide array of applications and coding styles.

4.2.3 Report Generator

The Report Generation and Output component is crucial for consolidating and presenting the results from the script's analysis. It is tasked with compiling a comprehensive report from the data procured by the Issue Identification and Reporting module, providing valuable insights into the identified coding violations, their specific locations within the Verilog code, and the details of the signals or variables involved.

Functionality:

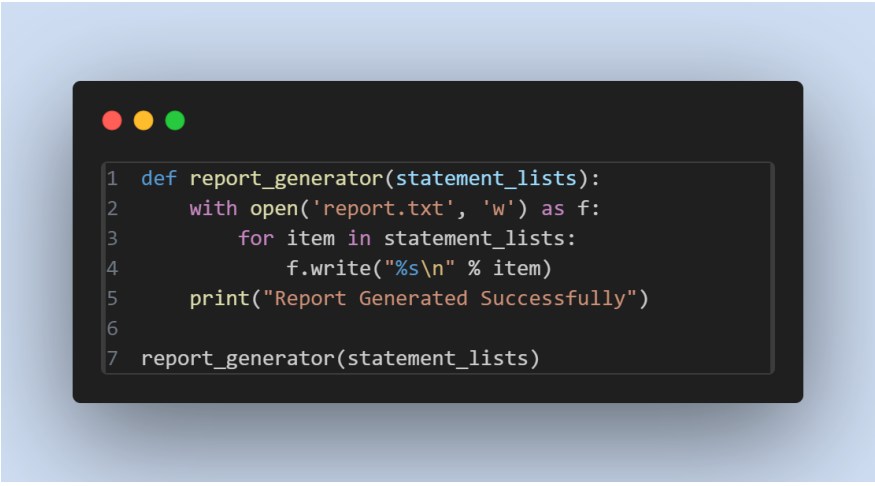
- **Report Compilation:** Aggregates and organizes the findings from the issue detection phase, preparing a detailed report of all identified issues.
- **Detailed Output:** Offers an exhaustive breakdown of each identified issue, including the nature of the violation, its precise location in the code, and any implicated signals or variables.
- **User-Friendly Formatting:** Ensures the results are formatted in a clear, understandable manner, making it easy for users to interpret and act upon the findings.

Report Contents:

- **Violation Names:** Enumerates the types of coding violations detected, providing a clear label for each issue.
- **Line Numbers:** ISpecifies the exact line numbers in the Verilog code where each violation was identified, aiding in quick location and review.
- **P**oints out the specific signals or variables that are affected by each identified violation, providing a comprehensive overview of the impact.

Accessibility:

- **Easily Accessible Reports:** Generates reports that are straightforward to access and interpret, ensuring users can effectively utilize the findings.
- **Integration with External Tools:** Facilitates the compatibility and integration of the reports with external tools for further analysis, visualization, or incorporation into a broader development environment.



```python
def report_generator(statement_lists):
    with open('report.txt', 'w') as f:
        for item in statement_lists:
            f.write("%s\n" % item)
    print("Report Generated Successfully")

report_generator(statement_lists)
```

4.3 Implementation Details

The script integrates the Verilog Parser, Issue Identification and Reporting, and Report Generation and Output components into a unified tool, creating a streamlined process that fits smoothly into the Verilog design workflow. Employing sophisticated parsing and analysis algorithms, the script ensures a thorough and accurate examination of the Verilog code for various potential issues. With its emphasis on flexibility and user-friendliness, the tool is designed to be accessible and valuable to users with different levels of expertise in Verilog design, offering customizable features to suit project-specific needs and preferences.

4.3.1 Utility Functions

```
module_counter = counting_modules(verilog_code)

module_lists_space = [[] for i in range
(module_counter)]

module_lists_space =
 creating_module_lists(verilog_code, module_lists
_space, module_counter)

line_num_list = []

def creating_line_num_list(module_lists_space):
    tmp = 0
    for i in module_lists_space:
        for line_num,j in enumerate(i):
            x = line_num+tmp
            line_num_list.append([module_lists_sp
ace.index(i),x,j])
# module number , line number , line
        tmp = x+1
    return line_num_list


def remove_comments_line_num_list(line_num_list):
    for i in line_num_list:
        if i[2].startswith('//'):
            line_num_list.remove(i)
        elif '//' in i[2]:
            index = i[2].find('//')
            tmp = i[2][index:].split(" ")
            if "synopsys" in tmp:
                break
            i[2] = i[2][:index]

```

```
line_num_list =
 creating_line_num_list(module_lists_space)
#remove_comments_line_num_list(line_num_list)
line_num_list

for i in line_num_list:
    if i[2].startswith('//'):
        line_num_list.remove(i)
    elif '//' in i[2]:
        index = i[2].find('//')
        tmp = i[2][index:].split(" ")
        if "synopsys" in tmp:
            break
        i[2] = i[2][:index]
line_num_list


# create lists according to module counter
module_lists = [[] for i in range
(module_counter)]
module_lists

# append each module to a module_lists from the y
module_lists =
 creating_module_lists(verilog_code, module_list
s, module_counter)

# remove all empty strings from the module_lists
module_lists = remove_empty_strings(module_lists)

module_lists = remove_comments(module_lists)

module_lists

```

Detailed Description

The provided code snippet involves a sequence of steps aimed at parsing and preprocessing Verilog code, organizing it into a structured format, and making it ready for further analysis. The main steps include:

Counting Modules (`counting_modules`)

- It starts by determining the number of modules in the provided Verilog code using the `counting_modules` function. The `module_counter` variable holds the count of modules found in `verilog_code`.

Initializing Module Lists (`module_lists_space`)

- It initializes `module_lists_space` with a list of empty lists, one for each module. This will later hold the lines of code pertaining to each module.

Creating Module Lists (`creating_module_lists`)

- Using the `creating_module_lists` function, it populates `module_lists_space` with the actual content. Each sublist in `module_lists_space` corresponds to one module from the Verilog code and contains all lines of code belonging to that module.

Creating Line Number List (`creating_line_num_list`)

- The `line_num_list` is initialized and populated using the `creating_line_num_list` function. It creates a comprehensive list where each item consists of the module number, line number, and the line content. This list is essential for tracking the location of specific lines of code within the original Verilog file, potentially for error reporting or further analysis.

Handling Comments

- While a function to remove comments from the `line_num_list` is provided (`remove_comments_line_num_list`), it's commented out in this snippet. However, the intention here is to clean up the code by removing comments to avoid any misinterpretation in later stages of analysis.

Initializing and Populating Final Module Lists (`module_lists`)

- It initializes `module_lists` with empty lists based on the number of modules. It then populates `module_lists` with the actual content, organizing the lines of code according to their respective modules. This is similar to `module_lists_space` but appears to be the finalized version after potential modifications or clean-up.

Removing Empty Strings and Comments

- The code cleans up `module_lists` by removing any empty strings and comments. This ensures that the lists only contain relevant and actionable lines of code for further processing.

In summary, the code is a series of steps designed to parse and preprocess Verilog code by organizing it into a structured format, cleaning it up, and making it ready for further analysis like detecting potential issues or understanding the structure. The end result is a set of lists where each list corresponds to a module and contains clean, comment-free code lines along with metadata like line numbers for easy reference. This structured format is crucial for any complex analysis or operations that might be performed on the Verilog code subsequently.

4.3.2 Uninitialized register

```python
def word_position_relative_to_equal(word, sentence):
    search_pattern = r'\b{}\b'.format(word)

    matches = [match for match in re.finditer(search_pattern, sentence)]

    if matches:
        word_index = matches[0].start()
        equal_index = sentence.find('=')

        if sentence.count('=') > 1:
            return 3  # not found

        last_word_index = matches[-1].start() + len(matches[-1].group()) - 1

        if equal_index == -1:
            return 3  # not found

        if word_index < equal_index and last_word_index > equal_index:
            return 2  # found both before and after equal

        elif word_index < equal_index:
            return 0  # before equal

        elif last_word_index > equal_index:
            return 1  # after equal
    else:
        return 3  # not found
```

Detailed Description

The function `word_position_relative_to_equal` is designed to analyze the position of a given word relative to the first equals sign '=' in a sentence (which is typically a line of code or expression). It returns a numeric code indicating the word's position, which is useful for parsing and analyzing code or expressions, particularly in contexts where the position of a variable or keyword relative to an assignment operator is significant. Here's how it works: **Initialization and Pattern Matching**

- The function accepts two parameters: `word`, the word to search for, and `sentence`, the sentence in which to search.
- It constructs a regex `search_pattern` to match the whole word boundary of the `word`.
- Then, it finds all matches of the word in the sentence using the regex pattern.

Analysis of Word Position Relative to '='

- If there are matches (i.e., the word is found in the sentence):
    - It gets the index of the first and last occurrence of the word.
    - It also finds the index of the first equals sign '=' in the sentence.

    * If there is more than one ‘=’ sign or no ‘=’ sign, it returns 3, indicating the word’s position is "not found" or indeterminate relative to ‘=’.
    * It then checks the relative position of the word to the ‘=’ sign:
        · If the word is entirely before ‘=’, it returns 0.
        · If the word is entirely after ‘=’, it returns 1.
        · If the word appears both before and after ‘=’, it returns 2, indicating the word spans across the ‘=’ sign.
- If the word is not found at all in the sentence, it returns 3, indicating "not found".

Return Values

- The function returns an integer code based on the word’s position:
    * 0: The word is entirely before the ‘=’ sign.
    * 1: The word is entirely after the ‘=’ sign.
    * 2: The word is both before and after the ‘=’ sign (spans across it).
    * 3: The word’s position is indeterminate relative to ‘=’, or the word is not found.

This function can be particularly useful in code analysis and transformation tools where understanding the role or position of variables or keywords in expressions is crucial. It can be used, for instance, to determine if a variable is being assigned a value or is used in an assignment to another variable, among other potential uses in syntactic analysis of code.

```python
def reg_list_for_unintialized(module_lists):
    reg_names = [] # list of lists first place is reg name and second place is module index
    for i in module_lists:
        for j in i:
            if j.startswith('reg') and not('=' in j):
                mask = 1
                if "]" in j:
                    j = j[j.find(']')+1:]
                    if "," in j:
                        j = j.replace(",","")
                j = j.replace("reg","")
                j = j.replace(";","")
                if j[0] == " ":
                    j = j[1:]
                if " " in j:
                    tmp = j.split(" ")
                    for t in tmp:
                        reg_names.append([t, module_lists.index(i)])
                else:
                    reg_names.append([j,module_lists.index(i)])
    return reg_names
```

Detailed Description

The `reg_list_for_uninitialized` function is designed to scan through a list of Verilog module definitions and identify registers that are declared but not initialized. Here's a detailed breakdown of its operations:

Initialization

The function initializes an empty list `reg_names`, which will store sublists containing register names and their corresponding module indices.

Iterating Through Modules and Lines

- The function accepts one parameter `module_lists`, which is expected to be a list of modules. Each module is itself a list of strings, each representing a line of Verilog code.
- It iterates through each module (`i`) and each line (`j`) within the module.

Identifying Registers

- For each line, it checks if the line starts with 'reg' and does not include an equals sign '=', indicating a declaration without initialization.
- If such a line is found, it enters a series of string manipulations:
    - If there are square brackets ']', indicating a bit-width declaration, it slices the string to remove this part and any following commas.
    - It then cleans up the line by removing the 'reg' keyword and any trailing semicolons.
    - Leading spaces are removed for cleanliness.
    - If there are spaces within what remains, it's assumed to be multiple register names declared in one line, so it splits these into individual names.

Storing Register Names and Module Indices

- For each identified register name, it creates a sublist containing the register name and the index of the module in which it was found.
- These sublists are appended to `reg_names` list.

Return Value

- Once all modules and lines have been processed, the function returns the `reg_names` list, now populated with sublists for each identified register. Each sublist contains a register name and the index of the module where that register is declared.

This function is particularly useful in the context of static code analysis for Verilog where it's important to identify registers that might not have been initialized, which can lead to unpredictable behavior in synthesized digital circuits. By identifying these registers, users can ensure that all parts of their hardware description are fully and explicitly defined before synthesis and simulation.

```python
def check_unintialized_reg(module_lists):
    mask = 0
    reg_names = reg_list_for_unintialized(module_lists)
#print(reg_names)
#print()
    for i in reg_names:
        module = module_lists[i[1]]
        #print(i[1])
        #print(i[0])
        mask = 0
        for j in module:
            if i[0] in j and not(j.startswith('case')):

                if word_position_relative_to_equal(i[0],j) == 0:
                    #print("before equal")
                    #print(j)
                    mask = 1
                elif word_position_relative_to_equal(i[0],j) == 1:
                    #print("after equal")
                    #print(j)
                    print("\nModule Name:", module[0])
                    for x in line_num_list:
                        if x[0] == i[1] and x[2] == j:
                            line_num = x[1] + 1
                    print("Line Number:", line_num)
                    statement_lists.append("\nModule Name: " + module[0])
                    statement_lists.append("Line Number : " + str(line_num))
                    print("Reg name: ", f"\"{i[0]}\"")
                    statement_lists.append("Reg name: " + f"\"{i[0]}\"")
                    print("Possible Uninitialized reg")
                    statement_lists.append("Possible Uninitialized reg")
                    print("====================================")
                    statement_lists.append("====================================")
                    #mask = 1
                elif word_position_relative_to_equal(i[0],j) == 2:
                    #print("both before and after equal")
                    #print(j)
                    print("\nModule Name:", module[0])
                    statement_lists.append("\nModule Name: " + module[0])
                    for x in line_num_list:
                        if x[0] == i[1] and x[2] == j:
                            line_num = x[1] + 1
                    print("Line Number:", line_num)
                    statement_lists.append("Line Number : " + str(line_num))
                    print("Reg name: ", f"\"{i[0]}\"")
                    statement_lists.append("Reg name: " + f"\"{i[0]}\"")
                    print("Possible Uninitialized reg")
                    statement_lists.append("Possible Uninitialized reg")
                    print("====================================")
                    statement_lists.append("====================================")
                    #mask = 1
                #print("--------------------")
            if mask == 1:
                break
```

```
            if j.startswith('case') and i[0] in j:
                #print(j)
                #print("case")
                print("\nModule Name:", module[0])
                statement_lists.append("\nModule Name: " + module[0])
                for x in line_num_list:
                    if x[0] == i[1] and x[2] == j:
                        line_num = x[1] + 1
                print("Line Number:", line_num)
                statement_lists.append("Line Number : " + str(line_num))
                print("Reg name: ", f"\"{i[0]}\"")
                statement_lists.append("Reg name: " + f"\"{i[0]}\"")
                print("Possible Uninitialized reg")
                statement_lists.append("Possible Uninitialized reg")
                print("====================================")
                statement_lists.append("====================================")
                break


check_unintialized_reg(module_lists)
```

Detailed Description

The `check_uninitialized_reg` function is a part of a larger codebase that analyzes Verilog modules to detect registers that might be uninitialized. It works by leveraging the `reg_list_for_uninitialized` function to get a list of registers that are declared but not initialized and then checks each register's usage in the module to determine if it's indeed uninitialized. Here's how the function operates:

Initialization

- It initializes a `mask` variable to 0 and retrieves a list of potentially uninitialized registers using the `reg_list_for_uninitia` function.

Iterating Through Registers

- For each register (and its module index) in `reg_names`, the function iterates through lines of the corresponding module:
    - If the register name is found in a line (and the line is not part of a 'case' statement), it checks the position of the word relative to the '=' character using the `word_position_relative_to_equal` function.
    - Depending on the returned value from `word_position_relative_to_equal`:
        - If the register is found before '=', the function assumes the register might be used and sets the `mask` to 1.
        - If the register is found after '=' (value 1) or both before and after '=' (value 2), the function considers it a possible uninitialized register and prints relevant information, including the module name, line number, and register name. This information is also added to a `statement_lists` for further reporting.

Printing and Recording Findings

- If a possible uninitialized register is detected, the function prints out details, including the module name, line number, and register name. This information is meant for debugging or reporting purposes and is appended to the `statement_lists` list for later use.

Handling 'Case' Statements

- If a line starts with 'case' and contains the register name, it is also considered a possible uninitialized register. The function prints out similar details for such cases.

Early Termination

- For each register, once an uninitialized scenario is detected (`mask` set to 1), or a 'case' statement is encountered with the register, the function breaks out of the inner loop to avoid redundant checks.

The function essentially provides a detailed report of registers that might be uninitialized in a given set of Verilog modules. This is crucial for digital design verification, as uninitialized registers can lead to undefined behaviors or bugs in synthesized circuits. By highlighting these potential issues, developers can take preemptive action to ensure all registers are properly initialized before simulation or synthesis.

4.3.3 Multiple Drivers

```python
def check_multidriven_variables_always_blocks(module_lists):
    modules_with_multidriven_variables = set()

    for module_index, module in enumerate(module_lists, start=1):
        # Always blocks and their contents
        always_blocks = []

        # Extract contents of always blocks
        inside_always = False
        current_always_block = []

        for line in module:
            if 'always' in line:
                inside_always = True
                current_always_block = ['always']
            elif inside_always:
                current_always_block.append(line.strip())
                if 'end' in line:
                    inside_always = False
                    always_blocks.append(current_always_block)

        # Compare always blocks to identify multidriven variables
        seen_variables = set()
        seen_variables_list = []
        for i, block1 in enumerate(always_blocks):
            for j, block2 in enumerate(always_blocks):
                if i != j:
                    # Check for 'variable ='
                    for line1 in block1[2:-1]:  # Skip 'always', 'begin', 'end'
                        for line2 in block2[2:-1]:  # Skip 'always', 'begin', 'end'
                            seen_variables_list.append(line2)
                            if '=' in line1 and '=' in line2:
                                variable_name1 = line1.split('=')[0].strip()
                                variable_name2 = line2.split('=')[0].strip()
                                if variable_name1 == variable_name2:
                                    seen_variables.add(variable_name1)

        # Print the results only if there are multidriven variables in the module
        if seen_variables and module_index not in modules_with_multidriven_variables:
            modules_with_multidriven_variables.add(module_index)
            print(f"\nModule {module_index}:", module[0])
            statement_lists.append(f"\nModule {module_index}: {module[0]}")
            for block in always_blocks:
                print("Always Block:", block)
                statement_lists.append("Always Block: " + str(block))
                for n in block:
                    for x in line_num_list:
                        for y in seen_variables_list:
                            if x[2] == n and y in n:
                                line_num = x[1] + 1
                                print("Line Number:", line_num)
                                statement_lists.append("Line Number : " + str(line_num))
                                break
                        pass

            print("Multidriven Variables:", seen_variables)
            statement_lists.append("Multidriven Variables: " + str(seen_variables))
            print("====================")
            statement_lists.append("====================")

# Example usage
check_multidriven_variables_always_blocks(module_lists)
```

Detailed Description

The `check_multidriven_variables_always_blocks` function is designed to analyze a list of Verilog modules to identify and report variables that are driven (assigned values) multiple times within different 'always' blocks. This is a common issue in digital logic design that can lead to unpredictable behavior or race conditions. Here's a breakdown of the function:

Initialization

- The function initializes a set `modules_with_multidriven_variables` to keep track of modules with multidriven variables. It iterates through each module in the `module_lists`, with `module_index` indicating the module's position in the list.

Identifying 'always' Blocks

- The function looks for 'always' blocks within each module. It identifies these blocks by the keyword 'always' and considers the subsequent lines until an 'end' is encountered as part of the block.
- Each block is stored in `always_blocks` as a list of its constituent lines.

Detecting Multidriven Variables

- For each 'always' block, the function compares its variables with those of every other block to identify variables that are assigned in multiple blocks.
- If a variable is assigned in one block (indicated by the presence of '='), and the same variable name appears on the left-hand side of '=' in another block, it's considered a multidriven variable and added to `seen_variables`.
- The function also maintains `seen_variables_list` to track the lines where these variables are found for later reporting.

Reporting Results

- If any multidriven variables are found in a module, the function prints out the module number and name, the details of each 'always' block, and the list of multidriven variables.
- For each always block and identified variable, it also aims to provide the specific line number within the original module where the variable is driven.
- It appends these details to `statement_lists` for further use or reporting.
- The function ensures each module with multidriven variables is only reported once by adding its index to `modules_with_multidriven_variables`.

Functionality Enhancements

- The function uses `line_num_list`, presumably a pre-generated list of line numbers and corresponding text from the modules, to match and report the exact line numbers of the multidriven variables within the Verilog code.

This function is crucial in the verification and analysis phase of hardware design, as multidriven variables can cause synthesis issues and unpredictable hardware behavior. By identifying such variables, developers can rectify the design to ensure a single source drives each variable within any given state or condition, adhering to good coding practices for hardware description languages like Verilog.

```python
def check_multidriven_variables_assign_statements(module_lists):
    for module_index, module in enumerate(module_lists, start=1):
        # Extracted variable names and sizes
        variables = set()
        # Assign statements and assigned variables
        assign_statements = []

        for line in module:
            # Check if the line contains an assign statement

            if 'assign' in line:
                parts = line.split()
                assign_index = parts.index('assign')

                # Extract the assigned variable name
                if assign_index < len(parts) - 1:
                    variable_name = parts[assign_index + 1].rstrip(';')
                    # Check if the variable is already assigned in another statement
                    if variable_name in variables:
                        assign_statements.append([line, variable_name])
                    else:
                        variables.add(variable_name)

        # Check for multidriven variables within the same module based on assign statements
        seen_variables = set()
        for statement, variable in assign_statements:
            # Check if the variable is repeated (multidriven) in the same module
            if variable in seen_variables:
                print(f"Module {module_index}: Variable '{variable}' is multidriven.")
            else:
                seen_variables.add(variable)

        # Print the results only if there are multidriven variables in the module
        if seen_variables:
            print(f"\nModule {module_index}:", module[0])
            statement_lists.append(f"\nModule {module_index}: {module[0]}")
            print("Assign Statements:", assign_statements)
            for n in assign_statements:
                for x in line_num_list:
                    if x[2] == n[0]:
                        line_num = x[1] + 1
                        print("Line Number:", line_num)
                        statement_lists.append("Line Number : " + str(line_num))
                        break


                pass
            statement_lists.append("Assign Statements: " + str(assign_statements))
            print("Multidriven Variables:", seen_variables)
            statement_lists.append("Multidriven Variables: " + str(seen_variables))
            print("====================")
            statement_lists.append("====================")

# Example usage
check_multidriven_variables_assign_statements(module_lists)
```

Detailed Description

The `check_multidriven_variables_assign_statements` function is designed to identify variables in Verilog modules that are driven multiple times across different 'assign' statements. This is a critical issue in digital design, as a variable should ideally have a single driving source to ensure predictable behavior. Here's how the function operates:

Initialization

- The function iterates through each module in the given `module_lists`, noting the `module_index`. It initializes:
  - A set `variables` to keep track of all variables encountered.
  - A list `assign_statements` to store details of assign statements and the variables they drive.

Iterating Through Module Lines

- The function examines each line in the module to identify 'assign' statements.
- If an 'assign' statement is found, it splits the line into parts and extracts the variable name immediately following the 'assign' keyword.
- It checks if this variable has already been assigned in another statement within the same module. If it has, it's considered a potential multidrive situation, and the line, along with the variable name, is added to `assign_statements`.

Detecting Multidriven Variables

- After processing all lines, the function iterates through `assign_statements` to identify any variables that appear more than once—indicating they are driven by multiple assign statements.
- Each identified multidriven variable is added to the `seen_variables` set.

Reporting Results

- If any multidriven variables are found, the function prints the module number and name, details of each relevant assign statement, and the list of multidriven variables.
- For each assign statement involved in multidriving a variable, the function also reports the specific line number within the original module, presumably using a pre-existing `line_num_list` that associates lines of code with their line numbers.
- These details are appended to `statement_lists` for later use or reporting, potentially in a summarized document or log.

End of Function

- The function concludes by potentially printing a list of all assign statements and multidriven variables for each module that contains such issues. It uses a clear demarcation ("==========") to separate reports for different modules.

By identifying and reporting variables that are driven by multiple 'assign' statements within the same module, this function helps in diagnosing potential conflicts or design flaws in Verilog code, encouraging cleaner and more reliable hardware descriptions. Such diagnostics are essential for ensuring the integrity and reliability of digital circuits before they are synthesized and implemented in hardware.

```python
def check_multidriven_variables_always_with_assign_statement(module_lists):
    modules_with_multidriven_variables = set()

    for module_index, module in enumerate(module_lists, start=1):
        # Always blocks and their contents
        always_blocks = []

        # Extract contents of always blocks
        inside_always = False
        current_always_block = []
        always_blocks_variables = []

        for line in module:
            if 'always' in line:
                inside_always = True
                current_always_block = ['always']
            elif inside_always:
                current_always_block.append(line.strip())
                # extract variable name from always block
                if '<=' in line :
                    variable_name = line.split('<=')[0].strip()
                    always_blocks_variables.append(variable_name)
                    break
                if '=' in line :
                    variable_name = line.split('=')[0].strip()
                    always_blocks_variables.append(variable_name)
                    break


        # Extracted variable names and sizes
        assign_variables = set()
        # Assign statements and assigned variables
        assign_statements = []

        for line in module:
            # Check if the line contains an assign statement
            if 'assign' in line:
                parts = line.split()
                assign_index = parts.index('assign')
                assign_statements.append([line, variable_name])


                # Extract the assigned variable name
                if assign_index < len(parts) - 1:
                    variable_name = parts[assign_index + 1].rstrip(';')
                    # Check if the variable is already assigned in another statement
                    if variable_name in assign_variables:
                        assign_statements.append([line, variable_name])
                    else:
                        assign_variables.add(variable_name)



        # check for multidriven variables within the same module based on always block and assign statements
        seen_variables = set()
        tmp = ""
        for variable in always_blocks_variables:
            # Check if the variable is repeated (multidriven) in the same module
            #print(assign_statements)
            if variable in assign_variables:
                seen_variables.add(variable)
                print(f"Module {module_index}: Variable '{variable}' is multidriven.")
                statement_lists.append(f"Module {module_index}: Variable '{variable}' is multidriven.")
                print("Assign Statements:", assign_statements)
                statement_lists.append("Assign Statements: " + str(assign_statements))
                for n in assign_statements:
                    for x in line_num_list:
                        if "assign" in x[2]:
                            # remove all except for variable name
                            tmp = x[2].split("=")
                            tmp = tmp[0].strip()
                            tmp = tmp.split(" ")
                            tmp = tmp[1]
                            #print(tmp)
                        if n[1] == tmp:
                            line_num = x[1] + 1
                            print("Line Number:", line_num)
                            statement_lists.append("Line Number : " + str(line_num))
                            break
                print("Multidriven Variables:", seen_variables)
                statement_lists.append("Multidriven Variables: " + str(seen_variables))
                print("======================================")
                statement_lists.append("======================================")
            else:
                assign_variables.add(variable)
```

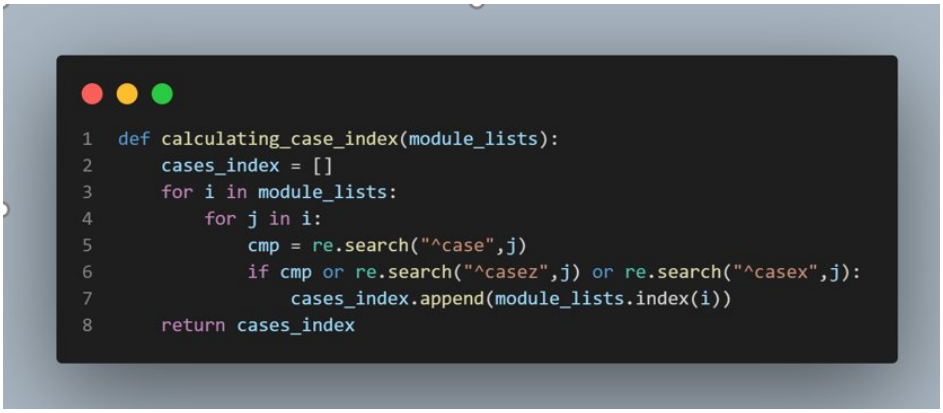
Detailed Description

The `check_multidriven_variables_always_with_assign_statement` function is designed to analyze Verilog modules to identify variables that are driven (assigned values) multiple times across different `always` blocks and `assign` statements. This condition is indicative of multidriven variables, which can lead to unpredictable behavior or race conditions in the hardware. Here's a breakdown of the function:

1. **Initialization**:
   - The function iterates through each module in the provided `module_lists`, identifying `always` blocks and `assign` statements within each module.
   - It initializes `modules_with_multidriven_variables` to keep track of modules with multidriven variables.
2. **Identifying `always` Blocks and Variables**:
   - It looks for `always` blocks within each module and extracts the variables being assigned within these blocks. Variables are identified by looking for '<=' or '=' operators and extracting the left-hand side as the variable name.
   - It collects all variables assigned in `always` blocks into `always_blocks_variables`.
3. **Identifying `assign` Statements and Variables**:
   - It also identifies all `assign` statements within the module and extracts the variables being assigned in these statements.
   - Each variable found in an `assign` statement is added to `assign_variables`.
4. **Detecting Multidriven Variables**:
   - After collecting all variables assigned in `always` blocks and `assign` statements, the function compares them to identify any variables that are driven by both an `always` block and an `assign` statement within the same module.
   - If a variable is found in both `always_blocks_variables` and `assign_variables`, it's considered multidriven, and its name is added to `seen_variables`.
5. **Reporting Results**:
   - If any multidriven variables are found, the function prints out details about the module, the variable, and relevant `assign` statements. It reports the module number, the variable name, and the line numbers of the relevant `assign` statements.
   - It also appends these details to `statement_lists` for further processing or reporting, potentially in a summarized document or log.
6. **Enhancements for Accuracy**:
   - The function attempts to accurately identify and report the line numbers of `assign` statements related to multidriven variables using a pre-existing `line_num_list` that associates lines of code with their line numbers.

By identifying and reporting variables that are driven by multiple sources within the same module, the `check_multidriven_variables_always_with_assign_statement` function helps in diagnosing potential conflicts or design flaws in Verilog code. This is essential for maintaining the reliability and predictability of digital circuits.

4.3.4 Non-Full Case checker



```python
def calculating_case_index(module_lists):
    cases_index = []
    for i in module_lists:
        for j in i:
            cmp = re.search("^case",j)
            if cmp or re.search("^casez",j) or re.search("^casex",j):
                cases_index.append(module_lists.index(i))
    return cases_index
```

Detailed Description

The `calculating_case_index` function is designed to scan through a list of Verilog modules and identify all the modules that contain `case`, `casex`, or `casez` statements. These statements are fundamental in Verilog for implementing decision control logic. Here's a breakdown of the function:

Initialization

- It begins with an empty list `cases_index`, which will hold the indices of modules containing any case statements.

Iterating Through Modules and Lines

- The function iterates through each module (`i`) within the `module_lists`. Each module is itself a list of strings, each string representing a line of Verilog code.
- For each line (`j`) in the module, it performs regex searches to check if the line starts with `case`, `casex`, or `casez`. The caret (`^`) in the regex pattern signifies the start of the line, ensuring that it only matches these keywords if they appear at the beginning of a line, which is typical for a Verilog case statement.

Identifying Modules with Case Statements

- If any of the regex searches (`cmp`, `casez`, `casex`) find a match, it means the current line is a case statement.
- The function then appends the index of the current module (`i`) to the `cases_index` list. This index is derived from the position of the module in the `module_lists`.

Return Value

- After iterating through all modules and lines, the function returns the `cases_index` list, which contains the indices of all modules that contain at least one case statement.

This function is particularly useful in the context of Verilog code analysis, where identifying modules with conditional logic is crucial for understanding the behavior, potential optimizations, and necessary debugging of digital circuits. By knowing which modules use case statements, users can focus their analysis, testing, or optimization efforts accordingly.

```python
def generating_reg_list(module_lists,any_specific_index):
    reg_list = []
    # in each list first number is case_index and second number is reg_index & size of reg is third number
    for i in any_specific_index:
        for j in module_lists[i]:
            if j.startswith('input'):
                j = j[len('input '):]
            if j.startswith('output'):
                j = j[len('output '):]
            if j.startswith('localparam'):
                j = j[len('localparam '):]
                if ']' in j or '[' in j:
                    low_bound = j.find('[')
                    high_bound = j.find(']')
                    size = j[low_bound+1:high_bound]
                    size = size.split(':')
                    size = int(size[0])-int(size[1])+1
                    j = j[high_bound+1:].replace(';', '')
                    j = j.replace(' ', '')
                    variables_names = j.split(',')
                    for var_name in variables_names:
                        reg_list.append([i, var_name, size])
                else:
                    size = 1
                    low_bound = j.find('g')
                    j = j[low_bound+1:].replace(';', '')
                    j = j.replace(' ', '')
                    variables_names = j.split(',')
                    for var_name in variables_names:
                        reg_list.append([i, var_name, size])

            if j.startswith('reg'):
                # store reg name & size in a list
                if ']' in j or '[' in j:
                    low_bound = j.find('[')
                    high_bound = j.find(']')
                    size = j[low_bound+1:high_bound]
                    size = size.split(':')
                    size = int(size[0])-int(size[1])+1
                    j = j[high_bound+1:].replace(';', '')
                    j = j.replace(' ', '')
                    variables_names = j.split(',')
                    for var_name in variables_names:
                        reg_list.append([i, var_name, size])
                else:
                    size = 1
                    low_bound = j.find('g')
                    j = j[low_bound+1:].replace(';', '')
                    j = j.replace(' ', '')
                    variables_names = j.split(',')
                    for var_name in variables_names:
                        reg_list.append([i, var_name, size])
    #print(reg_list)
    return reg_list
```

Detailed Description

The `generating_reg_list` function is designed to generate a list of registers (and their respective sizes) from specific modules within a list of Verilog modules. This function is particularly useful for gathering detailed information about the registers in a subset of modules for further analysis or processing. Here's how it works:

Initialization

– The function starts with an empty list `reg_list`, which will hold the details of registers identified in the specified modules.

Iterating Through Specified Modules and Lines

– The function accepts two parameters: `module_lists`, a list of modules where each module is a list of lines of Verilog code, and `any_specific_index`, which is a list of indices indicating the specific modules to be analyzed within `module_lists`.

- For each specified module index (`i` in `any_specific_index`), it iterates through each line (`j`) of the corresponding module in `module_lists`.

Identifying and Processing Register Declarations

- For each line in the module, the function looks for lines that start with 'input', 'output', 'localparam', or 'reg' as these are commonly used to declare registers or parameters in Verilog.
- Depending on the type of declaration, it processes the line to extract the variable names and their sizes.
- If the declaration includes a range (`[m:n]`), it calculates the size of the register based on this range.
- It then cleans up the line to extract individual variable names and appends a sublist for each register to `reg_list`. Each sublist contains the module index, the register name, and the size of the register.

Handling Different Declarations

- The function is designed to handle both single-bit (`reg a;`) and multi-bit (`reg [3:0] a;`) register declarations. It defaults to a size of 1 for single-bit registers.
- It similarly processes 'input', 'output', and 'localparam' declarations, assuming these might also specify the width of signals or parameters that can be relevant in the context of the function's usage.

Return Value

- After iterating through all specified modules and lines, the function returns the `reg_list`. Each element of this list is a sublist containing:
    - The index of the module where the register is found.
    - The name of the register.
    - The size of the register.

By generating a detailed list of registers and their sizes from specified modules, this function provides valuable information that can be used for various purposes, including but not limited to, analyzing the register usage, checking for potential issues, or simply documenting the register details of Verilog modules. It abstracts the complexity of parsing and understanding register declarations in Verilog, providing a straightforward and reusable tool for Verilog code analysis.

```
def check_full_case(module_lists):
    cases_index = calculating_case_index(module_lists)
    reg_list = generating_reg_list(module_lists,cases_index)
    size_list_for_case = []
    mask = 0
    for i in cases_index:
        for j in module_lists[i]:
            for t in reg_list:
                if t[0] == i and t[1] in j:
                    tmp = j.split("=")
                    if len(tmp) > 1:
                        #print(tmp)
                        pre_defined_reg = tmp[0]


            if j.startswith('case') or j.startswith('casez') or j.startswith('casex'):
                #print(module_lists[i][module_lists[i].index(j)+1:])
                line_after_case = module_lists[i][module_lists[i].index(j)+1:module_lists[i].index(j)+2]

                # convert from list to string
                line_after_case = ''.join(line_after_case)
                # remove all characters before :
                line_after_case = line_after_case[line_after_case.find(':')+1:]
                #print(line_after_case)
                try:
                    if pre_defined_reg in line_after_case:
                        break
                except:
                    pass
                if "synopsys" in j:
                    break
                bound = j.find('e')
                reg_name = j[bound+1:]
                reg_name = reg_name.replace(' ', '')
                reg_name = reg_name.replace('(', '')
                reg_name = reg_name.replace(')', '')
                reg_name = reg_name.replace(':', '')
                reg_name = reg_name
                #print(reg_name)
                # take size of reg_name
                for k in reg_list:
                    if k[1] == reg_name and k[0] == i:
                        size = k[2]
                        size_list_for_case.append(size)
                        break
                case_i = module_lists[i].index(j)

                rows_count = 0

                for line in module_lists[i][case_i+1:]:
                    if line.startswith('endcase'):
                        break
                    if line.startswith('default'):
                        mask = 1
                        break
                    if ":" in line:
                        rows_count += 1
                #print("rows_count", rows_count)
                if mask == 0:
                    if pow(2, size) != rows_count:
                        print("\nNon-Full Case:")
                        statement_lists.append("\nNon-Full Case")
                        print("Module", i + 1, ":", module_lists[i][0])
                        statement_lists.append("Module " + str(i + 1) + " : " + module_lists[i][0])
                        print(f"Size of reg \"{reg_name}\":", size)
                        statement_lists.append(f"Size of reg \"{reg_name}\" : " + str(size))
                        print("Number of variations:", rows_count)
                        statement_lists.append("Number of variations : " + str(rows_count))
                        print("Expected number of variations:", pow(2, size))
                        statement_lists.append("Expected number of variations : " + str(pow(2, size)))
                        print("Number of variations is not equal to expected number of variations")
                        statement_lists.append("Number of variations is not equal to expected number of variations")
                        print("====================================")
                        statement_lists.append("====================================")
                        print()
                        break
                    else:
                        break


#print(size_list_for_case)
check_full_case(module_lists)
```

Detailed Description

The `check_full_case` function is designed to verify that each `case`, `casex`, or `casez` statement within a list of Verilog modules has a full set of case items, thereby preventing incomplete case specifications which can lead to latches or unexpected behavior in hardware. Here's a detailed breakdown of how the function operates:

Initialization and Preprocessing

- The function starts by identifying all modules that contain case statements using `calculating_case_index` and generates a list of registers with `generating_reg_list` for the identified modules.
- It initializes `size_list_for_case` to keep track of sizes of the registers involved in the case statements and a `mask` to flag whether a default case is present.

Iterating Through Modules with Case Statements

- For each module index in `cases_index`, it iterates through each line of the module.
- It looks for lines starting with `case`, `casez`, or `casex` to identify the beginning of case statements.

Analyzing Case Statements

- For each identified case statement, it locates the register or signal that the case statement is evaluating and determines its size using information from the previously generated `reg_list`.
- It then counts the number of distinct case items (ignoring the default case) present in the case statement by counting the number of colons (`:`) that typically represent case item beginnings.
- It also checks if a 'default' case is provided by looking for the keyword 'default' within the case statement.

Determining Fullness of Case Statements

- For each case statement, the function calculates the expected number of variations based on the size of the register being evaluated (e.g., a 2-bit register would have 4 variations - 00, 01, 10, 11).
- It compares the count of actual case items with the expected number of variations. If these counts don't match and there's no 'default' case to cover missing scenarios, it identifies the case as "Non-Full" and notes this as an issue.

Reporting Non-Full Case Statements

- If a non-full case statement is found, the function prints detailed information about it, including the module number, the line number of the case statement, the name and size of the register, the actual and expected number of variations, and a message indicating that the number of variations doesn't match the expected number.
- All this information is also appended to `statement_lists` for further processing or output.

Utility of the Function

- By ensuring that each case statement has a comprehensive set of case items or a default case, the function helps in avoiding incomplete specifications that might lead to unpredictable hardware behavior. This is crucial for reliable digital design and synthesis from Verilog code.

This function is part of a larger suite of tools intended for static analysis of Verilog code, helping designers ensure their hardware description is robust, complete, and free of common logical errors before proceeding to synthesis and implementation.

4.3.5 Non-Parallel Case checker

```python
def extract_text_before_colon(input_text):
    # Find the index of the colon
    colon_index = input_text.find(':')

    if colon_index != -1:
        # Extract the text before the colon
        text_before_colon = input_text[:colon_index].strip()
        return text_before_colon
    else:
        return "None"
```

Detailed Description

The `extract_text_before_colon` function is designed to parse a given string and extract the portion of text that occurs before the first colon (`:`) character. This function is particularly useful in parsing and analyzing structured text or code where colons might separate key-value pairs, definitions, or other significant segments. Here's how the function operates:

Finding the Colon

- The function begins by searching for the index of the first colon in the `input_text` using the `find` method. The `find` method returns the lowest index of the substring (in this case, the colon) if found, otherwise -1.

Extracting Text

- If a colon is found (i.e., `colon_index` is not -1), the function proceeds to extract all text before this colon. It does so by slicing the `input_text` from the start up to the `colon_index`.
- It then applies the `strip` method to remove any leading or trailing whitespace from the extracted text. This cleanup is particularly useful in ensuring that the returned text is neat and does not include extra spaces that might have been around the colon in the original text.

Return Value

- If the text before the colon is successfully extracted, it returns this text.
- If no colon is found in the `input_text`, the function returns "None".

By providing a clean and straightforward way to get the text before a colon in a string, this function can be a valuable component in parsing and interpreting structured data or code, such as configuration settings, programming language structures, or any other colon-separated format. Its implementation is concise and efficient, suitable for integration into larger text processing or analysis routines.

```python
def can_be_number(input_text):
    try:
        input_text = input_text.replace(' ', '')
        if input_text[2] == 'b':
            input_text = input_text.split('b')[1]
            #print(input_text)
        elif input_text[2] == 'd':
            input_text = input_text.split('d')[1]
        elif input_text[2] == 'h':
            input_text = input_text.split('h')[1]
            input_text = int(input_text, 16)
        int(input_text)
        return True
    except :
        return False
can_be_number("1'b0")
```

Detailed Description

The `can_be_number` function is designed to check if a given input text string represents a numerical value in various potential formats, particularly those commonly used in hardware description languages like Verilog. It specifically looks for binary (`b`), decimal (`d`), or hexadecimal (`h`) number representations. Here's how the function operates:

Preprocessing

- The function begins by removing all spaces from the `input_text` to ensure that extra spaces do not affect the interpretation of the number.

Checking for Numerical Formats

- The function then checks if the third character of the `input_text` indicates a binary (`b`), decimal (`d`), or hexadecimal (`h`) number. This is based on a common notation where the number is preceded by its size and type, for example, "1'b0" for a 1-bit binary 0 or "2'hF" for a 2-bit hexadecimal F.
- Depending on the type indicated, it splits the string at the respective character and takes the part after it as the number to be interpreted. For hexadecimal numbers, it also converts the extracted part to an integer using base 16.

Validity Check

- After isolating the potential number, it attempts to convert the string to an integer using `int(input_text)`. If this conversion is successful, it means the text can be interpreted as a number, and the function returns True.
- If any error occurs during the process (caught by the `except` block), such as if the input text is not a valid number or doesn't follow the expected format, the function returns False.

Return Value

- The function returns True if the `input_text` can be interpreted as a binary, decimal, or hexadecimal number. Otherwise, it returns False.

By testing the input "1'b0", which is a common way to represent a single binary 0 in hardware description languages, the function returns True, confirming that it correctly interprets this as a binary number. This function can be particularly useful in parsing and validating numbers in configurations, specifications, or any scenario where numbers may come in various representational formats.

```python
def is_parallel_sequence(lst):
    lst0 = [element.replace('?', '0') for element in lst]
    lst0 = [element.replace('x', '0') for element in lst0]
    lst1 = [element.replace('?', '1') for element in lst]
    lst1 = [element.replace('x', '1') for element in lst1]

    set0 = set(lst0)
    set1 = set(lst1)
    if len(set0) < len(lst0) or len(set1) < len(lst1):
        return False
    else:
        return True
```

Detailed Description

The `is_parallel_sequence` function is designed to determine if a given list of binary sequences (with potential wildcards) can be considered parallel. In the context of hardware description languages like Verilog, parallel case statements should cover all possible combinations of inputs without overlap, ensuring that each case is unique. This function aims to validate that by interpreting wildcard characters ('?' or 'x') as both '0' and '1'. Here's how the function works:

Preprocessing Lists

- The function creates two new lists from the original list (`lst`): `lst0` and `lst1`. Both lists are versions of the original, where wildcard characters '?' and 'x' are replaced with '0' and '1', respectively.
- `lst0` is the version where all wildcards are pessimistically interpreted as '0', and `lst1` is where they are optimistically interpreted as '1'.

Creating Sets for Uniqueness Check

- It converts both `lst0` and `lst1` into sets, `set0` and `set1`. Sets in Python are collections of unique elements, so any duplicates in `lst0` or `lst1` are removed in this conversion. This step is crucial for checking if the original list had any overlapping or redundant elements.

Checking for Parallelism

- The function compares the lengths of `set0` and `set1` against the original `lst0` and `lst1`. If any duplicates were removed (indicating the presence of non-unique elements), the length of the set would be less than the length of the original list.
- If either `set0` or `set1` is shorter than its corresponding list, it means there were non-unique elements in the original list, implying that the sequence is not parallel (as it contains overlapping cases). In this case, the function returns False.
- If the lengths match for both sets and lists, indicating all elements were unique and there's no overlap, the function returns True, implying that the sequence is parallel.

Return Value

- The function returns True if the provided list can be considered a parallel sequence, meaning it doesn't contain overlapping or redundant elements when interpreting wildcards as both '0' and '1'. Otherwise, it returns False.

This function can be particularly useful in validating the integrity of case statements in Verilog or similar scenarios where ensuring non-overlapping, comprehensive coverage of cases is critical. By checking for parallelism, designers can avoid logical errors or inefficiencies in their digital circuits.

```python
def check_parallel_case(module_lists):
    parallel_case = calculating_case_index(module_lists)
    mask2 = 0
    for i in parallel_case:
        mask = 0
        case_values = []
        for j in module_lists[i]:

            k_counter = 0
            parallel_case_counter = 0
            if j.startswith('case') or j.startswith('casez') or j.startswith('casex'):
            # iterate over lines after case
                #print(j)
                case_number = j
                if "synopsys" in j:
                    tmp = j.split(" ")
                    if "parallel_case" in tmp:
                        mask2 = 1
                        break


                for k in module_lists[i][module_lists[i].index(j)+1:]:
                    #print(k)
                    k_counter += 1
                    num = extract_text_before_colon(k)
                    if num != "None" and num != "default":
                        case_values.append(num)
                    if can_be_number(num):
                        parallel_case_counter += 1
                        #print("yes")
                    # mask = 1

                    if k.startswith('endcase'):
                        break

        for t in case_values:
            if can_be_number(t):
                mask = 1
            else:
                mask = 0
                break

        if mask == 0 and mask2 == 0 :
            if is_parallel_sequence(case_values) == False:
                print("Non-Parallel Case:")
                statement_lists.append("\nNon-Parallel Case:")
                print("Module", i + 1, ":", module_lists[i][0])
                statement_lists.append("Module " + str(i + 1) + " : " + module_lists[i][0])
                for x in line_num_list:
                    if x[0] == i and x[2] == case_number:
                        line_num = x[1] + 1
                print("Line Number:", line_num)
                statement_lists.append("Line Number : " + str(line_num))
                print("====================================")
                statement_lists.append("====================================")
                print()

        if mask2 == 1:
            mask2 = 0
```

Detailed Description

The `check_parallel_case` function is designed to analyze a list of Verilog modules and identify non-parallel case statements. Parallel case statements are important in digital design because they ensure that for each possible input, exactly one case is executed, preventing ambiguity and potential synthesis issues. Here's how the function works:

Initialization and Case Identification

- It starts by identifying all modules with case statements using the `calculating_case_index` function and initializes `mask2` to track specific conditions (like the presence of a "synopsys parallel_case" directive).
- It iterates through each identified module index in `parallel_case`.

Iterating Through Modules and Lines

- For each module, it iterates through the lines to find case statements, indicated by lines starting with 'case', 'casez', or 'casex'.
- Once it finds a case statement, it checks if it's marked as a "parallel_case" by Synopsys pragmas/directives. If so, it skips further checks for this case statement (`mask2` is set to 1).

Collecting Case Values

- For each case statement not skipped, it iterates through the lines following the case keyword to collect all case values (the part before the colon) using the `extract_text_before_colon` function. It ignores any 'default' labels and non-numeric values.
- These case values are stored in `case_values`, which will be used to determine if the case statement is parallel.

Checking for Parallelism

- After collecting all case values for a particular case statement, the function checks if these values form a parallel sequence using the `is_parallel_sequence` function. This involves replacing any wildcard characters and checking for unique entries.
- If the sequence is not parallel (indicating overlapping or incomplete case items) and there's no "parallel_case" directive (`mask2` is 0), the function identifies it as a "Non-Parallel Case".

Reporting Non-Parallel Cases

- For any non-parallel case statements found, the function prints and records details, including the module number, line number, and a message indicating it's a non-parallel case. This information is appended to `statement_lists` for future use or reporting.

Function Utility

- This function is crucial for verifying the integrity and efficiency of case statements in Verilog modules. Ensuring that case statements are parallel helps avoid logic errors and hardware synthesis problems, making the verification process an integral part of the design and debugging cycle in hardware description.

By automating the detection of non-parallel cases, the `check_parallel_case` function aids developers in creating more reliable and deterministic digital circuits, ensuring that every possible input is accounted for exactly once in the design's conditional logic.

4.3.6 Arithmetic Overflow

```python
def checkArithmeticOverflow(module_lists):
    # extracting the variables from the module
    variable_list = [[] for _ in range(len(module_lists))]

    for module_index, module in enumerate(module_lists, start=1):
        #print("Module Number:", module_index)
        for variable_declaration in module:
            # check if the variable is input, output, wire, or reg
            if variable_declaration.startswith(('input ', 'output ', 'wire ', 'reg ')):
                # Extract variable name and size
                parts = variable_declaration.split()
                parts[-1] =  parts[-1].rstrip(';')
                # Variable names are strings after '[number:number]' or after 'reg', 'wire', 'input', 'output'
                variable_names = [part.strip(',') for part in parts[1:] if part not in ('reg', 'wire', 'input', 'output')]
                for i in variable_names:
                    if '[' in i and ']' in i:
                        variable_names.remove(i)


                index = 0
                for i in variable_names:
                    if '=' in i:
                        variable_names.pop(index)
                        variable_names.pop(index)
                    index += 1
```

```python
        variable_size = 1  # Default size is 1

        # Check if [number-1:0] pattern is present
        if '[' in variable_declaration and ']' in variable_declaration:
            size_part = variable_declaration.split('[')[1].split(']')[0]

                # Extract the size correctly
            if ':' in size_part:
                sizes = size_part.split(':')
                variable_size = abs(int(sizes[0]) - int(sizes[1])) + 1
            else:
                variable_size = int(size_part) + 1
        # Store the variable names and size
        variable_list[module_index - 1].extend([[name, variable_size] for name in variable_names if name])

#print("Variable List:", variable_list[module_index - 1])
```

```
# extracting the operations from the module
operation_list = []
for operation in module:
    if '=' in operation:
        if '+' in operation or '-' in operation or '*' in operation or '/' in operation:
            # Extract the operation
            parts = operation.split('=')
            parts[-1] = parts[-1].rstrip(';')
            parts = [part.split(' ') for part in parts]
            # Remove empty strings
            for part in parts:
                while '' in part:
                    part.remove('')

            print("Parts:", parts)

            left_side_size = 0
            right_side_size = 0
            for variable in variable_list[module_index - 1]:
                if variable[0] in parts[0]:
                    left_side_size = variable[1]
                    break
            for variable in variable_list[module_index - 1]:
                if variable[0] in parts[1]:
                    right_side_size = max(variable[1], right_side_size)

            binary_size = re.findall(r'\b(\d+\'b[01]+)\b', operation)
            for i in binary_size:
                size = i.split('\'')[0]
                size = int(size)
                if size > right_side_size:
                    right_side_size = size




            if left_side_size <= right_side_size:
                print("\nPossible Arithmetic Overflow in module", module_index, ":", module[0])
                statement_lists.append("\nPossible Arithmetic Overflow in module " + str(module_index) + " : " + module[0])
                for x in line_num_list:
                    if x[0] == module_index-1 and x[2] == operation:
                        line_num = x[1] + 1
                print("Line Number:", line_num)
                statement_lists.append("Line Number : " + str(line_num))
                print("Line: ", operation)
                statement_lists.append("Line: " + operation)
                print("Left side size:", left_side_size)
                statement_lists.append("Left side size: " + str(left_side_size))
                print("Right side size:", right_side_size)
                statement_lists.append("Right side size: " + str(right_side_size))




            operation_list.append(parts)


if len(operation_list) == 0:
    pass
else:
    print("Operation List:", operation_list)
    statement_lists.append("Operation List: " + str(operation_list))
    print("====================")
    statement_lists.append("====================")

#print()
```

Detailed Description

The `checkArithmeticOverflow` function is designed to analyze arithmetic operations within a list of Verilog modules for potential overflow issues. It identifies situations where the size of variables on the left-hand side of an assignment may not be sufficient to hold the result of the operation on the right-hand side. Here's a detailed breakdown of the function:

Initialization

– The function initializes `variable_list` to store variable names and sizes for each module. It then iterates through each module in the provided `module_lists`, identifying variables and their sizes based on declarations like 'input', 'output', 'wire', and 'reg'.

Variable Extraction

- For each variable declaration in the module, it extracts the variable name and size, handling both single-bit and multi-bit declarations. It accounts for the typical Verilog bit-width specification format (`[m:n]`) to calculate the size.

Identifying Arithmetic Operations

- The function then iterates through each line in the module to find lines with arithmetic operations (`+`, `-`, `*`, `/`). When it finds such a line, it splits it into parts to identify the variables involved and the operation itself.
- It also accounts for binary constants in operations, extracting their sizes directly from the operation line to ensure that the right-hand side size calculation is accurate.

Size Comparisons and Overflow Check

- For each arithmetic operation, the function compares the size of the variable on the left-hand side of the assignment (`left_side_size`) with the calculated size of the entire operation on the right-hand side (`right_side_size`).
- If the left-side size is less than or equal to the right-side size, there's a potential for arithmetic overflow, and the function prints details about the possible overflow, including the module number, line number, the line itself, and the sizes of the left and right sides.

Reporting Potential Overflows

- All instances of potential arithmetic overflow are printed out with relevant information for further analysis or debugging. This information is also appended to `statement_lists`, presumably for generating a comprehensive report after the function execution.

Utility of the Function

- The function is beneficial in verifying the integrity of arithmetic operations in Verilog code, particularly for preventing overflow issues that could lead to unexpected behavior or logic errors in synthesized digital circuits. By ensuring that variables are properly sized to handle the results of operations, developers can create more robust and reliable hardware designs.

By automating the detection of potential arithmetic overflow issues, `checkArithmeticOverflow` aids developers in preemptively addressing one of the common pitfalls in hardware description and ensuring that the designed circuits will behave as expected under all legal input conditions.

4.3.7 Inferring latch

```python
def check_infer_latch(module_lists):
    line_count = 0

    for module_index, module in enumerate(module_lists, start=1):
        always_blocks = []
        used_signals = set()
        module_declaration_line = module[0]
        used_signals.update(set(re.findall(r'\b(\w+)\b', re.search(r'\bmodule\s+\w+\s*\((.*?)\);', module_declaration_line).group(1))))
        for i, line in enumerate(module):
            if re.search(r'always\s*@', line):
                always_blocks.append(i)


        for always_index in always_blocks:
            sensitivity_line = module[always_index]
            sensitivity_line = sensitivity_line.replace("always", "").replace("@", "").strip()

            # Extract the block content including the line with 'always' keyword
            block_content = [sensitivity_line] + module[always_index + 1:]

            # Check for latch inference scenarios
            check_sensitivity_list(sensitivity_line, module_index, line_count + always_index + 1, used_signals)
            check_feedback_loop(block_content, module_index, line_count + always_index + 1)
            check_if_without_else(block_content, module_index, line_count + always_index + 1)
            check_case_without_default(block_content, module_index, line_count + always_index + 1)

        # Update line_count for the next module
        line_count += len(module)
```

Detailed Description

The `check_infer_latch` function is designed to analyze Verilog modules for scenarios that might infer latches. Latches are often unintended in synchronous designs and can lead to issues with timing and behavior. They are typically inferred due to incomplete sensitivity lists or conditional statements in `always` blocks. Here's how the function operates:

*Initialization

- The function iterates through each module provided in `module_lists`, keeping track of the total line count (`line_count`) across all modules for accurate line number reporting.

Extracting Used Signals

- For each module, it extracts the signals used in the module declaration (i.e., the module's ports) using regex and adds them to the `used_signals` set. This is essential for checking sensitivity lists in `always` blocks later.

Identifying `always` Blocks

- The function searches for `always` blocks within each module. It identifies these blocks by lines starting with the keyword `always` followed by `@`, indicating the sensitivity list of the block.

Analyzing `always` Blocks

- For each identified `always` block, the function:
    - Extracts the sensitivity list, cleaning it up to remove the keywords `always` and `@`.
    - Constructs the block content, including the sensitivity list and all subsequent lines until the end of the block.

- It then checks for several common scenarios that might infer latches:
    - **Sensitivity List Issues:** Using `check_sensitivity_list`, it checks if the sensitivity list is incomplete or incorrect, which could lead to latches as the block might not trigger correctly for all relevant signal changes.
    - **Feedback Loops:** Using `check_feedback_loop`, it checks for combinational feedback loops, which can inadvertently create latches.
    - **Missing 'else' in Conditional Statements:** Using `check_if_without_else`, it checks for `if` statements without corresponding `else` parts. Incomplete conditional statements are a common source of inferred latches.
    - **Case Statements Without Default:** Using `check_case_without_default`, it checks for `case` statements without a `default` case, another potential source of inferred latches.

Utility and Reporting

- By systematically checking each `always` block in every module for these conditions, the function aims to identify and report specific lines and constructs that might infer latches. This is crucial for designers aiming for synchronous, latch-free designs, especially in FPGA and ASIC implementations where timing predictability and control are paramount.

Updating Line Count

- After analyzing each module, it updates the `line_count` to include the lines in the current module, ensuring that line numbers reported for subsequent modules are accurate.

This function is a part of a larger suite of analysis tools aimed at improving the quality and predictability of hardware designs by identifying and warning about constructs that might infer latches. Inferred latches can be a significant source of design issues, making this type of analysis extremely valuable in the verification and validation stages of digital design.

```python
def check_sensitivity_list(sensitivity_line, module_index, line_number, used_signals):
    # Check if sensitivity_line is "@*" or contains clk
    if "*" in sensitivity_line or "clk" in sensitivity_line:
        return

    # Extract signals from the sensitivity line
    sensitivity_list = re.findall(r'\b([a-zA-Z_]\w*)\b', sensitivity_line)
    # Remove non-signal elements from the sensitivity list
    sensitivity_list = [signal for signal in sensitivity_list if signal not in ["*", "("]]

    # Check for missing signals
    missing_signals = set()
    for signal in used_signals:
        if signal not in sensitivity_list:
            missing_signals.add(signal)

    for x in line_num_list:
        if sensitivity_line in x[2]:
            line_number = x[1] + 1

    # Print results
    if missing_signals:
        print(f"\nMay Infer Latch in module {module_index}, : {module_lists[module_index-1][0]} , line: {line_number}")
        statement_lists.append(f"May Infer Latch in module {module_index}, : {module_lists[module_index-1][0]} , line: {line_number}")
        print(f"Reason: Signal(s) missing in the sensitivity list: {', '.join(missing_signals)}")
        statement_lists.append(f"Reason: Signal(s) missing in the sensitivity list: {', '.join(missing_signals)}")
        print("====================================")
        statement_lists.append("====================================")
        return
```

Detailed Description

The `check_sensitivity_list` function is part of a larger set of functions aimed at analyzing Verilog modules for potential issues that might infer latches. Specifically, this function examines the sensitivity list of an `always` block to ensure that it includes all signals used in the block. An incomplete sensitivity list is a common source of unintended latches in synthesizable Verilog. Here's a breakdown of its operation:

Early Return for Wildcard Sensitivity Lists

– The function first checks if the sensitivity list is either a wildcard list (indicated by `@*`) or contains the clock signal (`clk`). In either case, it is assumed to be comprehensive, and the function returns early as there's no need to check for missing signals.

Extracting Signals from the Sensitivity List

– It uses a regex to extract all word-like strings from the sensitivity list, presumed to be signal names. It then filters out any non-signal elements (like the wildcard character `*` or parenthesis `(`).

Checking for Missing Signals

– The function compares the extracted signals against the `used_signals`, which presumably contains all signals used within the block. It identifies any signals from `used_signals` that are not in the sensitivity list and adds them to `missing_signals`.

Updating Line Number

– For accurate reporting, the function attempts to update the `line_number` to correspond to the line number of the sensitivity list within the entire module, presumably using a pre-existing list of lines and numbers (`line_num_list`).

Reporting Missing Signals

- If any `missing_signals` are found, the function prints and records information indicating that the module and specific line might infer a latch. The report includes the module number, module name, line number, and the missing signals. This information is critical for debugging and rectifying the potential latch issue.

Utility and Importance

- Sensitivity lists are crucial in defining when an `always` block should execute. A missing signal in the sensitivity list can result in an inferred latch, as the synthesis tool assumes the block should hold its value until the next trigger. Therefore, ensuring a complete sensitivity list is vital for predictable synchronous design.
- This function aids in the verification and debugging process by automating the detection of such sensitivity list omissions, guiding designers to create more robust and reliable digital circuits.

By systematically checking the sensitivity lists of `always` blocks, `check_sensitivity_list` helps prevent one of the common mistakes in Verilog coding that leads to latches, thereby enhancing the quality and predictability of the hardware design.

```python
def check_if_without_else(block_content, module_index, line_number):
    found_if = False
    found_else = False

    for line in block_content:
            if re.search(r'\bif\b', line):
                for x in line_num_list:
                    if line in x[2]:
                        line_number = x[1] + 1
                found_if = True

            # Check for 'else' inside 'always' block
            if found_if and re.search(r'\belse\b', line):
                found_else = True



    # If we reach here, it means 'if' was not followed by 'else' inside 'always' block
    if found_if and not found_else:
        print(f"\nInfer Latch in module {module_index}, : {module_lists[module_index - 1][0]}, line: {line_number}")
        statement_lists.append(f"Infer Latch in module {module_index}, : {module_lists[module_index - 1][0]}, line: {line_number}")
        print("Reason: 'if' statement without 'else' detected")
        statement_lists.append("Reason: 'if' statement without 'else' detected")
        print("====================================")
        statement_lists.append("====================================")
```

Detailed Description

The `check_if_without_else` function is designed to analyze the contents of an `always` block in a Verilog module to detect any `if` statements that are not properly paired with `else` statements. Missing `else` clauses in conditional statements can lead to unintended latches, as the synthesis tool may infer that the value should be held (latched) when the condition is not met. Here's how the function operates:

Initialization

– It initializes two flags: `found_if` to indicate whether an `if` statement has been found and `found_else` to indicate whether a corresponding `else` statement has been found.

Iterating Through Block Content

– The function iterates through each line in `block_content`, which contains the lines of an `always` block extracted from a Verilog module.

Identifying `if` and `else` Statements

– For each line, it uses regex to search for the presence of `if` and `else` statements.

– If an `if` statement is found, `found_if` is set to True, and it attempts to update the `line_number` to correspond to the line number of the `if` statement within the entire module, using a pre-existing `line_num_list`.

– It then continues to search the following lines for an `else` statement corresponding to the `if`. If an `else` is found after the `if`, `found_else` is set to True.

Checking for Missing `else`

– After iterating through all lines, it checks if an `if` was found without a corresponding `else`. This is indicated by `found_if` being True and `found_else` being False.

Reporting Inferred Latch

- If an `if` without `else` is detected, the function prints and records information indicating a potential inferred latch. It includes the module index, module name, line number of the `if` statement, and a message noting the absence of an `else`. This information is appended to `statement_lists` for further reporting or analysis.

Utility and Importance

- Ensuring every `if` statement in an `always` block has a corresponding `else` is critical for defining behavior for all possible input combinations and preventing synthesis tools from inferring latches. This function helps identify such issues, guiding developers to create more deterministic and latch-free designs.
- By automating the detection of these issues, `check_if_without_else` contributes to the verification and validation processes in digital circuit design, improving the reliability and predictability of the resulting hardware.

By systematically checking for missing `else` statements in conditional constructs, this function assists in preempting one of the common pitfalls in hardware description languages, helping to ensure that the resulting hardware design is robust and functions as intended.

```python
def check_case_without_default(block_content, module_index, line_number):
    found_case = False
    found_default = False

    for line in block_content:
        if re.search(r'^\s*case\b', line):
            if not re.search(r'//\s*synopsys\s*full_case', line):
                for x in line_num_list:
                    if line in x[2]:
                        line_number = x[1] + 1
                found_case = True

        if found_case and re.search(r'^\s*default\b', line):
                found_default = True

    if found_case and not found_default:
        print(f"\nMay Infer Latch in module {module_index},: {module_lists[module_index-1][0]}, line: {line_number}")
        statement_lists.append(f"May Infer Latch in module {module_index},: {module_lists[module_index-1][0]}, line: {line_number}")
        print("Reason: 'case' statement without 'default' detected")
        statement_lists.append("Reason: 'case' statement without 'default' detected")
        print("====================================")
        statement_lists.append("====================================")
        return
```

Detailed Description

The `check_case_without_default` function is specifically aimed at detecting 'case' statements within an `always` block of Verilog modules that lack a 'default' case. In Verilog, omitting the 'default' case in a 'case' statement might lead to latch inference if not all possible case scenarios are explicitly covered. Here's how it works:

Initialization

– The function initializes two boolean flags, `found_case` and `found_default`, to track whether a 'case' statement and its corresponding 'default' clause are found within the block content.

Iterating Through Block Content

– The function iterates through each line in `block_content`, which includes the lines of an `always` block extracted from a Verilog module.

Identifying 'case' Statements

– For each line, it uses regex to search for a 'case' statement (lines starting with 'case' and not commented out as a 'synopsys full_case'). If found, it sets `found_case` to True and attempts to update the `line_number` using `line_num_list`.

Searching for 'default' Clause

– Once a 'case' statement is found, the function continues to look for a 'default' clause in the subsequent lines within the same 'case' block. If a 'default' is found, `found_default` is set to True.

Detecting Cases Without 'default'

– After iterating through all lines, if a 'case' statement (`found_case` is True) without a corresponding 'default' (`found_default` is False) is detected, it indicates a potential issue where the 'case' statement might infer a latch.

Reporting Potential Latch Inference

- If such a 'case' without 'default' is found, the function prints and records information indicating a potential latch inference. It includes the module index, module name, line number of the 'case' statement, and a message highlighting the absence of 'default'. This information is appended to `statement_lists` for further reporting or analysis.

Utility and Importance

- Ensuring every 'case' statement in Verilog has a corresponding 'default' clause is crucial for defining behavior for all possible input combinations and preventing synthesis tools from inferring latches. This function helps identify such issues, guiding developers to create more deterministic and latch-free designs.
- By automating the detection of these issues, `check_case_without_default` contributes to the verification and validation processes in digital circuit design, improving the reliability and predictability of the resulting hardware.

By methodically checking for missing 'default' clauses in 'case' statements, this function assists in preempting one of the common pitfalls in hardware description languages, ensuring that the resulting hardware design functions as intended without unintended latches or undefined behaviors.

```python
def check_feedback_loop(block_content, module_index, line_number):
    dependencies = {}  # Dictionary to store signal dependencies

    # Extract signal dependencies from the block content
    for line in block_content[1:]:  # Exclude the sensitivity line
       match = re.search(r'\b(\w+)\s*(<=|=)\s*(.+?)\s*;', line)
       if match:
            left_signal, operator, right_expr = match.groups()
            dependencies[left_signal] = re.findall(r'\b\w+\b', right_expr)

    # Check for feedback loops
    for signal in dependencies:
        if signal in dependencies[signal]:
            for x in line_num_list:
                if block_content[0] in x[2]:
                    line_number = x[1] + 1
            print(f"\nMay Infer Latch in module {module_index}, : {module_lists[module_index-1][0]}, line: {line_number}")
            statement_lists.append(f"\nMay Infer Latch in module {module_index}, : {module_lists[module_index-1][0]}, line: {line_number}")
            print("Reason: Combinational Feedback loop detected")
            statement_lists.append("Reason: Combinational Feedback loop detected")
            print("====================================")
            statement_lists.append("====================================")
            return


# Call the modified checker function
check_infer_latch(module_lists)
```

Detailed Description

The `check_feedback_loop` function is part of a suite of functions designed to detect potential latch inference in Verilog modules, specifically by identifying feedback loops within `always` blocks. Feedback loops in combinational logic can inadvertently create latches or result in unstable behavior. Here's how the function operates:

Initialization

– It starts by creating an empty dictionary `dependencies` to track signal dependencies within the `always` block. Each key will be a signal name, and the corresponding value will be a list of signals that it depends on.

Extracting Signal Dependencies

– The function iterates through each line in `block_content` (excluding the sensitivity list, the first line). It uses regex to identify assignments (= or <=) and extracts the left-hand signal (the one being assigned) and the right-hand expression (the dependency).

– It then further extracts any word-like strings from the right-hand expression, interpreting them as signal names the left-hand signal depends on. These are stored in the `dependencies` dictionary under the respective left-hand signal name.

Checking for Feedback Loops

– The function iterates through the `dependencies` dictionary to check for each signal if it depends directly on itself, which would indicate a feedback loop. This check is performed by seeing if the signal's own name appears in its list of dependencies.

– If a feedback loop is found, it means the signal's value is being used in the computation of its next value without any intervening storage element, which is a classic scenario for unintended latch inference.

Updating Line Number and Reporting

- For accurate reporting, the function attempts to update the `line_number` to correspond to the line number of the detected feedback loop within the entire module, using a pre-existing `line_num_list`.
- If a feedback loop is detected, the function prints and records a message indicating potential latch inference due to a feedback loop, including the module index, module name, and the line number. This information is critical for debugging and rectifying the potential latch issue.

Utility and Importance

- Detecting feedback loops in combinational logic is essential for ensuring stable and predictable circuit behavior. Feedback loops can result in the synthesis tool inferring latches or causing the design to behave in unexpected ways. Identifying and resolving such loops is vital in digital design, especially for synchronous designs targeting FPGAs or ASICs.
- By automating the detection of feedback loops, `check_feedback_loop` aids in the verification and validation process of digital circuits, enhancing the reliability and predictability of the resulting hardware.

The `check_feedback_loop` function is an example of how static analysis can be used to detect and prevent common pitfalls in hardware design, contributing to the creation of more robust and well-behaved digital systems.

4.3.8 Unreached Block

```python
def extract_initial_values(module_lists):
    initial_values = {}

# Improved regex pattern to capture Verilog style binary values
    initial_pattern = re.compile(r'\b(\w+)\s*=\s*(\d+\'b[01]+);')
    inside_initial_block = False
    for lines in module_lists:
        for line in lines:
            if 'initial' in line:
                inside_initial_block = True
            elif 'end' in line and inside_initial_block:
                inside_initial_block = False
            elif inside_initial_block:
                match = initial_pattern.search(line)
                if match:
                    var, value = match.groups()
                    initial_values[var] = value
    return initial_values
```

Detailed Description

The function `extract_initial_values` is designed to parse through a list of Verilog modules and extract initial values set in the `initial` blocks. These initial values are commonly used in Verilog to set up initial conditions for simulation or specify default values for registers. Here's a breakdown of its operations:

Initialization

- It initializes an empty dictionary `initial_values` to hold the variable names and their initial values.

Defining a Regex Pattern

- It sets up an `initial_pattern` regex pattern designed to match Verilog-style binary value assignments (e.g., `var_name = 4'b1010;`). This pattern is used to find and extract both the variable name and its initial value.

Iterating Through Modules and Lines

- The function iterates through each module and each line within those modules in `module_lists`. It looks for `initial` blocks, which are standard in Verilog for defining initial conditions.

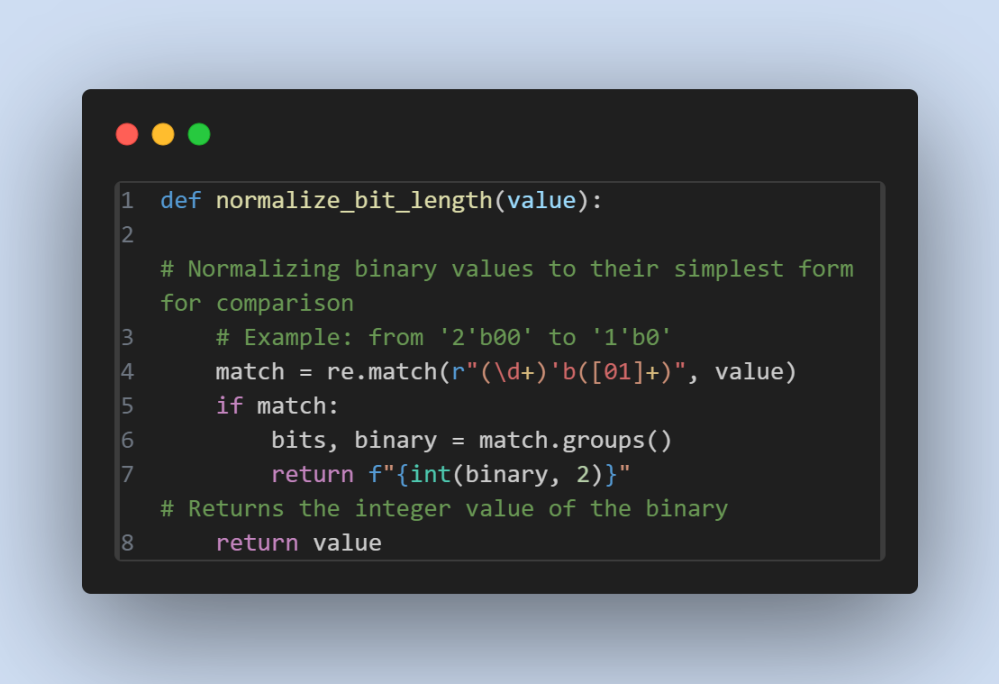
Detecting and Parsing `initial` Blocks

- When it finds an `initial` keyword, it marks that it's inside an initial block (`inside_initial_block = True`). It stays in this mode until it encounters an `end` keyword, signifying the end of the `initial` block.
- For every line inside an `initial` block, it applies the `initial_pattern` regex to find and extract variable assignments. When a match is found, it captures the variable name and value, adding them to the `initial_values` dictionary.

Return Value

- After processing all modules, the function returns the `initial_values` dictionary, which contains the variable names and their extracted initial values as key-value pairs.

This function is particularly useful for simulation setups or analyses that need to understand or modify the initial state of various variables in a Verilog design. By extracting initial values, the function aids in the preparation or understanding of how a Verilog module will behave when first run or reset, providing valuable insights into the design's intended initial state.



```python
def normalize_bit_length(value):

    # Normalizing binary values to their simplest form for comparison
    # Example: from '2'b00' to '1'b0'
    match = re.match(r"(\d+)'b([01]+)", value)
    if match:
        bits, binary = match.groups()
        return f"{int(binary, 2)}"
    # Returns the integer value of the binary
    return value
```

Detailed Description

The function `normalize_bit_length` is designed to convert binary representations in Verilog (which may include explicitly stated bit lengths) to their simplest integer form. This function is useful when you need a consistent format for comparing or further processing binary values, typically when those values are provided in Verilog's bit-length specific notation. Here's how it operates:

Regex Matching

- The function uses a regular expression to match and capture the bit length and binary value from the provided `value` string. The expected format is something like "4'b1010", where "4" is the number of bits and "1010" is the binary number.

Normalization

- If a match is found, it splits the value into two parts: `bits` (the bit length) and `binary` (the actual binary number).
- It then converts the binary part into its integer representation using `int(binary, 2)`. This conversion strips off any leading zeros and ignores the specified bit length, effectively normalizing the value.

Return Value

If the provided `value` is in the expected binary format, it returns the integer representation of the binary number. Also, If no match is found (meaning the `value` is not in the expected format), it simply returns the original `value`.

This function is particularly helpful in contexts where binary values are represented with varying bit lengths, but you need to compare or compute based on their numerical value regardless of how many bits were initially specified. By converting all binary values to their simplest integer form, `normalize_bit_length` ensures consistency and simplifies further operations or comparisons.

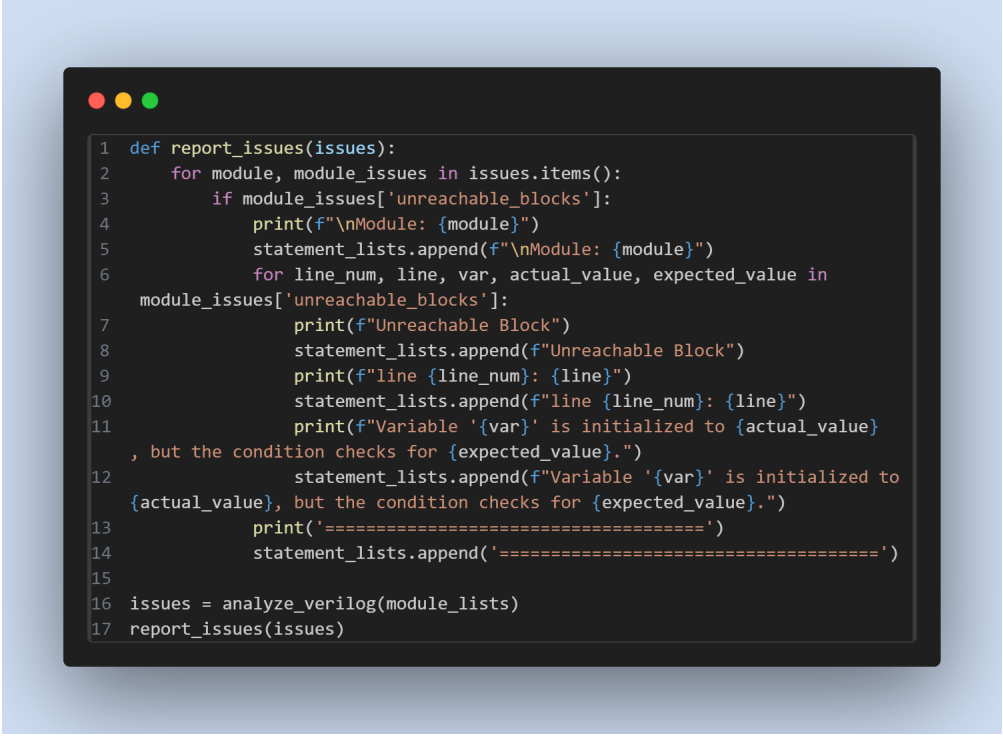
```python
def analyze_verilog(module_lists):
    issues = {}
    current_module = None
    line_num = 0

    initial_values = extract_initial_values(module_lists)

    if_condition_pattern = re.compile(r'if\s*\(\s*(\w+)\s*==\s*(\d+\'b[01]+)\s*\)')
    for lines in module_lists:
        for line in lines:
            for x in line_num_list:
                if x[2] == line:
                    line_num = x[1] + 1
            match = re.search(r'module\s+(\w+)', line)
            if match:
                current_module = match.group(1)
                issues[current_module] = {
                    'unreachable_blocks': [],
                    'module_line_num': line_num,
                    'initial_values': initial_values.copy()
                }
                continue

            if not current_module:
                continue

            match = if_condition_pattern.search(line)
            if match:
                var, expected_value = match.groups()
                actual_value = issues[current_module]['initial_values'].get(var)

                # Normalizing the bit-length for comparison
                if actual_value and normalize_bit_length(actual_value) != normalize_bit_length(expected_value):
                    issues[current_module]['unreachable_blocks'].append((line_num, line, var, actual_value, expected_value))

    return issues
```

Detailed Description

The `analyze_verilog` function is designed to scan through a list of Verilog modules and identify potential issues, specifically focusing on unreachable code blocks due to initial value mismatches in `if` conditions. Here's a breakdown of its operation:

1. **Initialization**:
   - The function initializes an empty dictionary `issues` to hold potential issues detected in each module, categorized by module name.
   - It also initializes `current_module` to keep track of the module currently being analyzed and `line_num` for maintaining the line number.
2. **Extracting Initial Values**:
   - It calls `extract_initial_values` to get a dictionary of initial values for variables across all modules.
3. **Iterating Through Modules and Lines**:
   - The function iterates through each line in each module in `module_lists`. It updates the line number (`line_num`) based on a pre-existing `line_num_list` that associates lines of code with their line numbers.
4. **Identifying Modules and Tracking Issues**:
   - For each line, it uses regex to check if the line marks the beginning of a new module (`module+(+)`). If so, it updates `current_module` and initializes an entry in `issues` for this module.
   - Each module's entry in `issues` includes a list for 'unreachable_blocks', the 'module_line_num', and a copy of 'initial_values'.
5. **Detecting Unreachable Blocks**:
   - The function searches for `if` conditions that compare a variable to a binary value (using `if_condition_pattern`).
   - If such a condition is found, it retrieves the variable's initial value from the `initial_values` dictionary.
   - It then normalizes both the expected value (from the `if` condition) and the actual initial value using `normalize_bit_length` and compares them.
   - If the normalized initial value does not match the normalized expected value in the `if` condition, it indicates that the block of code within the `if` condition is potentially unreachable. Such instances are added to the 'unreachable_blocks' list for the current module in `issues`.
6. **Return Value**:
   - After processing all lines in all modules, the function returns the `issues` dictionary. For each module, this includes any detected 'unreachable_blocks' with details such as the line number, the problematic line of code, the variable involved, its actual initial value, and the expected value.

By systematically checking for discrepancies between initial values and conditional checks in the code, the `analyze_verilog` function helps identify blocks of code that might never execute, aiding in the optimization and debugging of Verilog modules. This is crucial for ensuring the reliability and efficiency of digital circuits described by the Verilog code, especially before proceeding to synthesis and implementation.



```python
def report_issues(issues):
    for module, module_issues in issues.items():
        if module_issues['unreachable_blocks']:
            print(f"\nModule: {module}")
            statement_lists.append(f"\nModule: {module}")
            for line_num, line, var, actual_value, expected_value in module_issues['unreachable_blocks']:
                print(f"Unreachable Block")
                statement_lists.append(f"Unreachable Block")
                print(f"line {line_num}: {line}")
                statement_lists.append(f"line {line_num}: {line}")
                print(f"Variable '{var}' is initialized to {actual_value}, but the condition checks for {expected_value}.")
                statement_lists.append(f"Variable '{var}' is initialized to {actual_value}, but the condition checks for {expected_value}.")
            print('====================================')
            statement_lists.append('====================================')

issues = analyze_verilog(module_lists)
report_issues(issues)
```

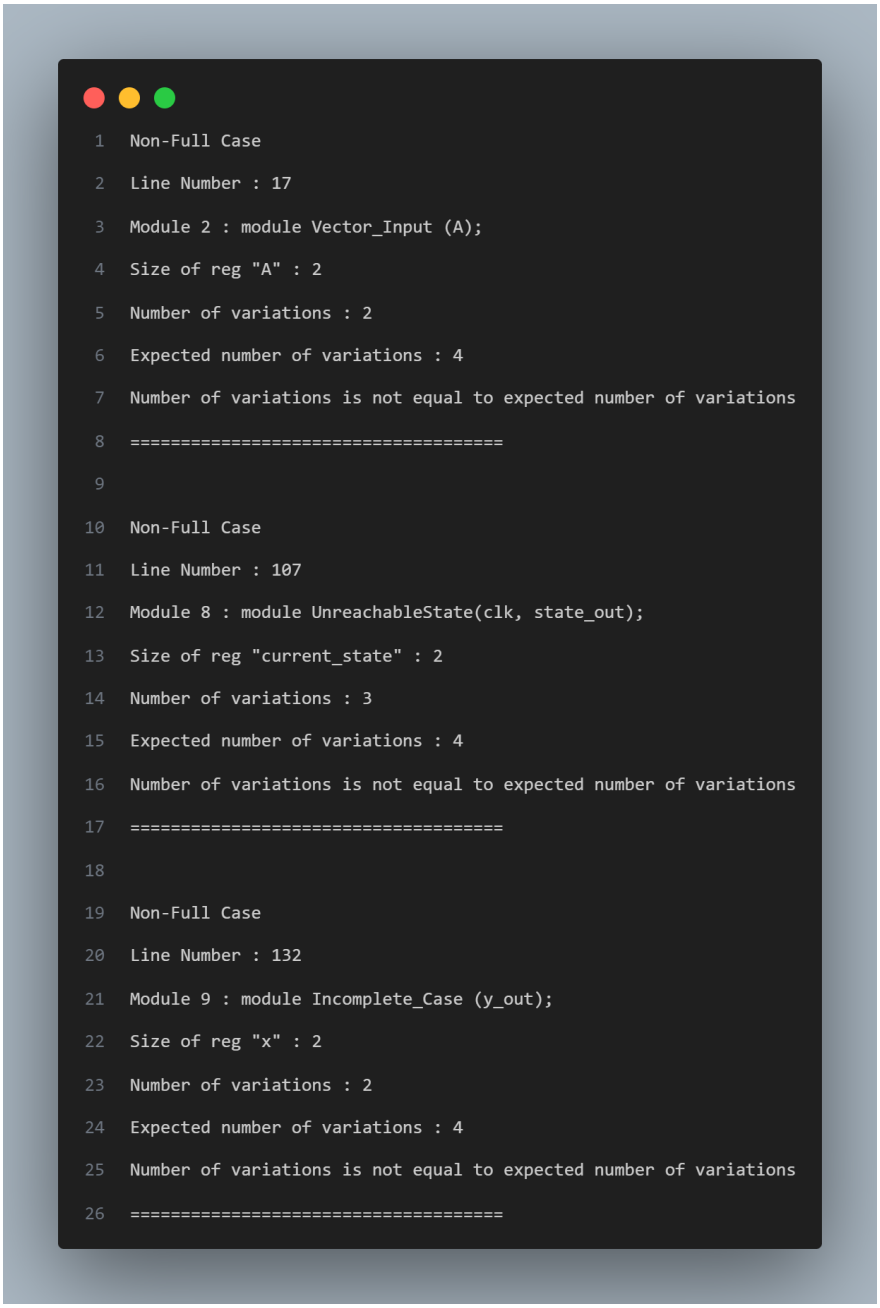
Detailed Description

The `report_issues` function takes a dictionary of issues identified in Verilog modules, likely generated by the `analyze_verilog` function, and prints out a detailed report for each module where issues were detected. Here's how it operates:

1. **Iterating Through Issues**:
   - The function iterates through each module in the `issues` dictionary. Each module's issues are accessed via `module_issues`.
2. **Checking for Unreachable Blocks**:
   - For each module, it checks if there are any 'unreachable_blocks' identified. These are blocks of code that might never execute due to discrepancies between initial values and conditional checks.
3. **Reporting Details**:
   - If unreachable blocks are found, it prints and records the module name. Then, for each unreachable block, it provides details including:
     * The line number where the issue was detected.
     * The line of code itself.
     * The variable involved and its actual initial value versus the expected value in the condition.
   - This information helps in pinpointing exactly where and why the issue occurs, aiding in debugging and optimizing the Verilog code.
4. **Formatting the Report**:
   - It neatly formats the report with clear separation and headers for each module and each issue within the module. This formatting is mirrored in both the printed output and the entries appended to `statement_lists`.

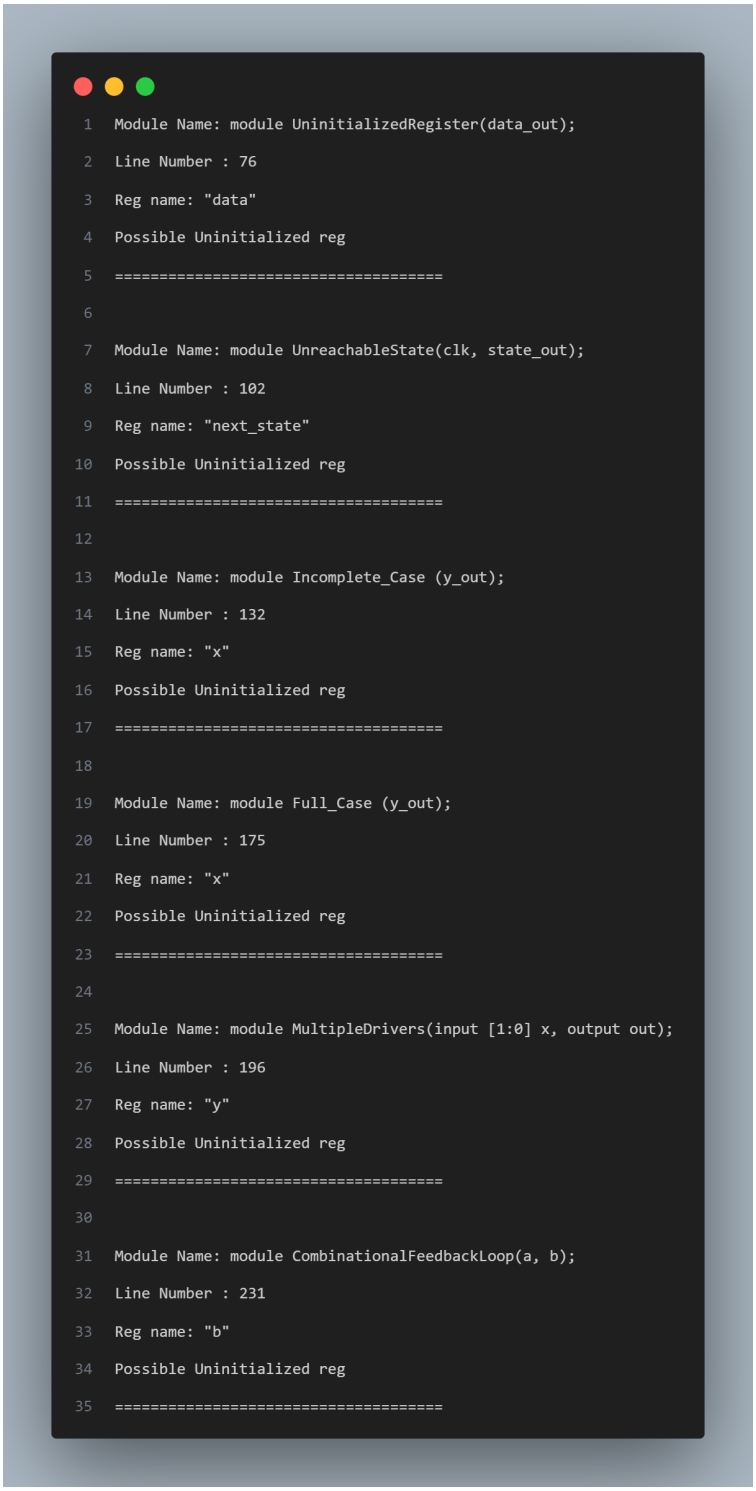
5 Results Generated

5.1 Non-Full Case



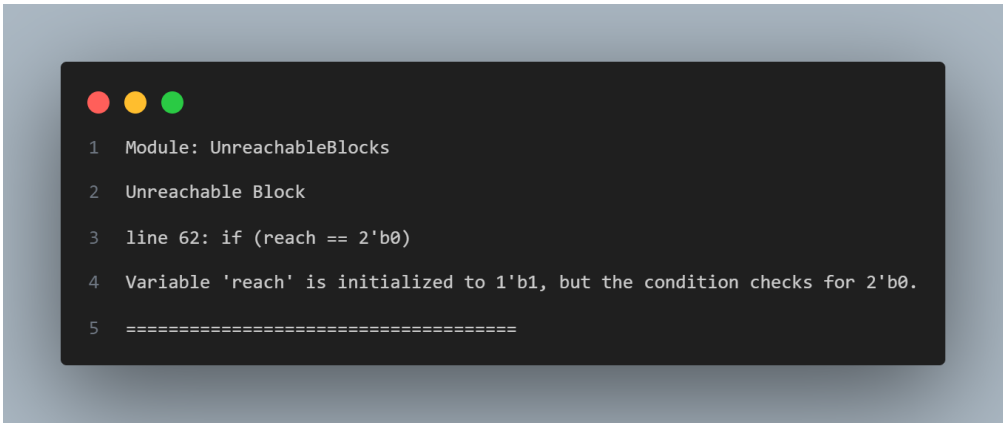
```
Non-Full Case
Line Number : 17
Module 2 : module Vector_Input (A);
Size of reg "A" : 2
Number of variations : 2
Expected number of variations : 4
Number of variations is not equal to expected number of variations
====================================

Non-Full Case
Line Number : 107
Module 8 : module UnreachableState(clk, state_out);
Size of reg "current_state" : 2
Number of variations : 3
Expected number of variations : 4
Number of variations is not equal to expected number of variations
====================================

Non-Full Case
Line Number : 132
Module 9 : module Incomplete_Case (y_out);
Size of reg "x" : 2
Number of variations : 2
Expected number of variations : 4
Number of variations is not equal to expected number of variations
====================================
```

5.2 Uninitialized Register

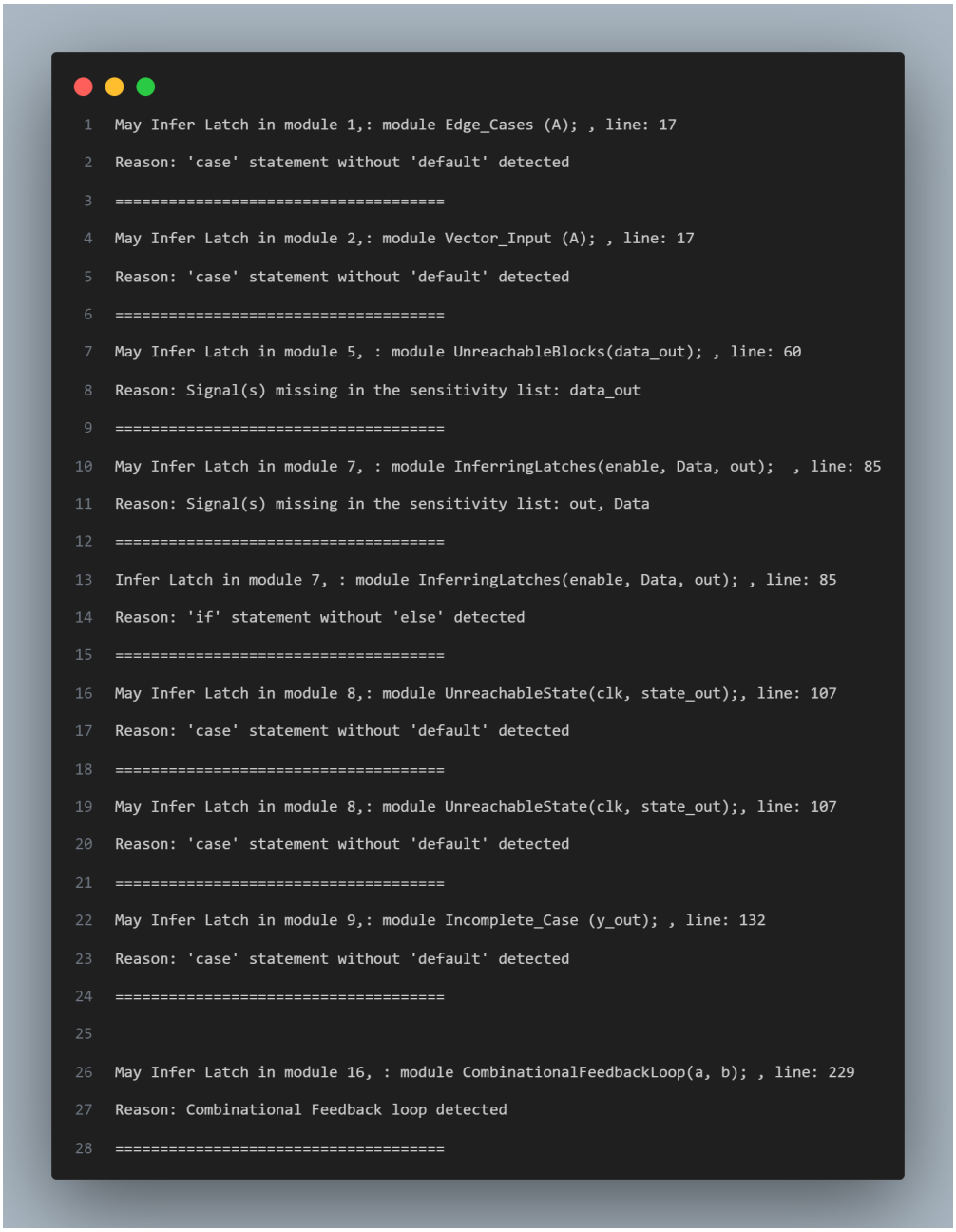


```
Module Name: module UninitializedRegister(data_out);
Line Number : 76
Reg name: "data"
Possible Uninitialized reg
====================================

Module Name: module UnreachableState(clk, state_out);
Line Number : 102
Reg name: "next_state"
Possible Uninitialized reg
====================================

Module Name: module Incomplete_Case (y_out);
Line Number : 132
Reg name: "x"
Possible Uninitialized reg
====================================

Module Name: module Full_Case (y_out);
Line Number : 175
Reg name: "x"
Possible Uninitialized reg
====================================

Module Name: module MultipleDrivers(input [1:0] x, output out);
Line Number : 196
Reg name: "y"
Possible Uninitialized reg
====================================

Module Name: module CombinationalFeedbackLoop(a, b);
Line Number : 231
Reg name: "b"
Possible Uninitialized reg
====================================
```

5.3 Unreachable Block

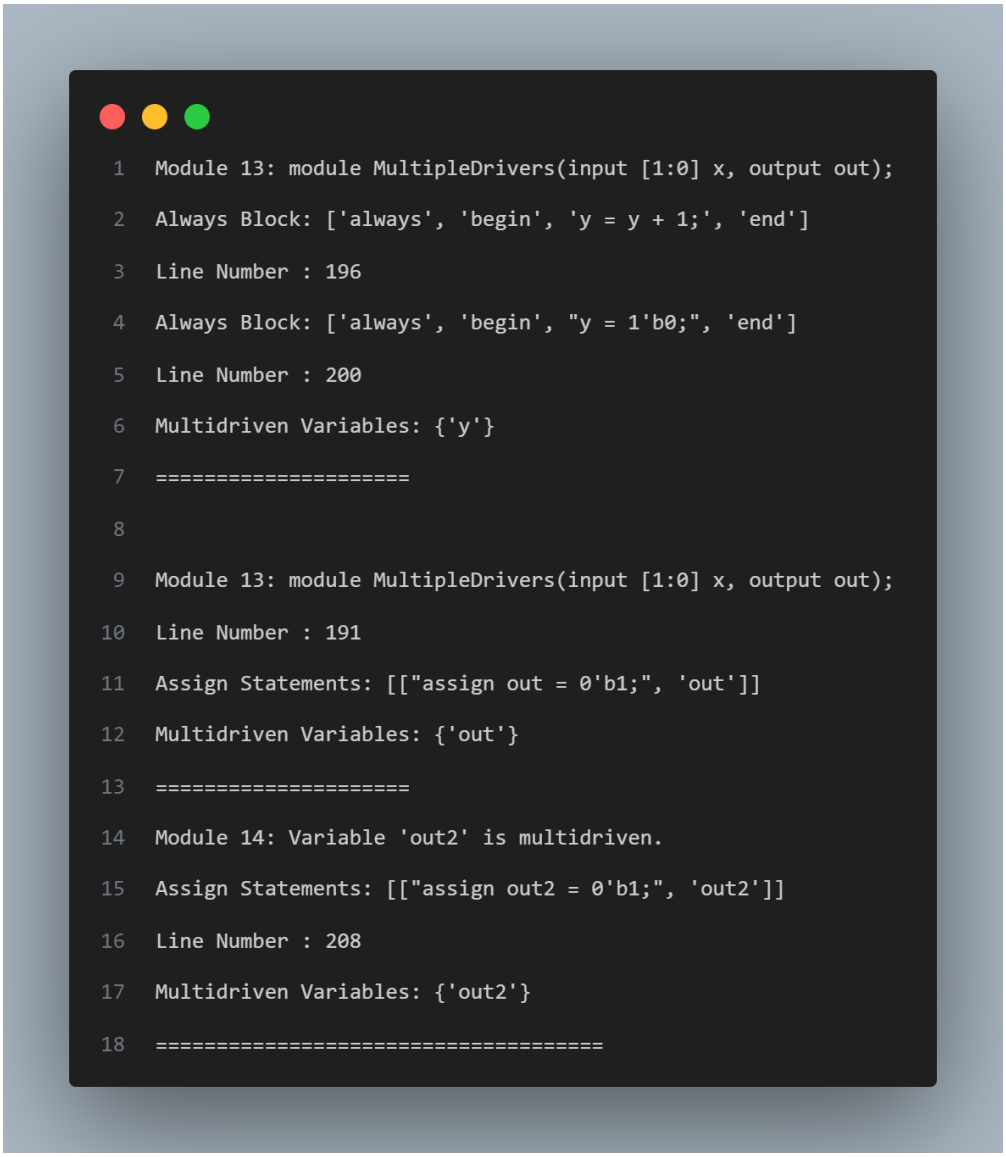


5.4 Inferring Latches



```
May Infer Latch in module 1,: module Edge_Cases (A); , line: 17
Reason: 'case' statement without 'default' detected
====================================
May Infer Latch in module 2,: module Vector_Input (A); , line: 17
Reason: 'case' statement without 'default' detected
====================================
May Infer Latch in module 5, : module UnreachableBlocks(data_out); , line: 60
Reason: Signal(s) missing in the sensitivity list: data_out
====================================
May Infer Latch in module 7, : module InferringLatches(enable, Data, out);  , line: 85
Reason: Signal(s) missing in the sensitivity list: out, Data
====================================
Infer Latch in module 7, : module InferringLatches(enable, Data, out); , line: 85
Reason: 'if' statement without 'else' detected
====================================
May Infer Latch in module 8,: module UnreachableState(clk, state_out);, line: 107
Reason: 'case' statement without 'default' detected
====================================
May Infer Latch in module 8,: module UnreachableState(clk, state_out);, line: 107
Reason: 'case' statement without 'default' detected
====================================
May Infer Latch in module 9,: module Incomplete_Case (y_out); , line: 132
Reason: 'case' statement without 'default' detected
====================================

May Infer Latch in module 16, : module CombinationalFeedbackLoop(a, b); , line: 229
Reason: Combinational Feedback loop detected
====================================
```

5.5 Multi driven Variables

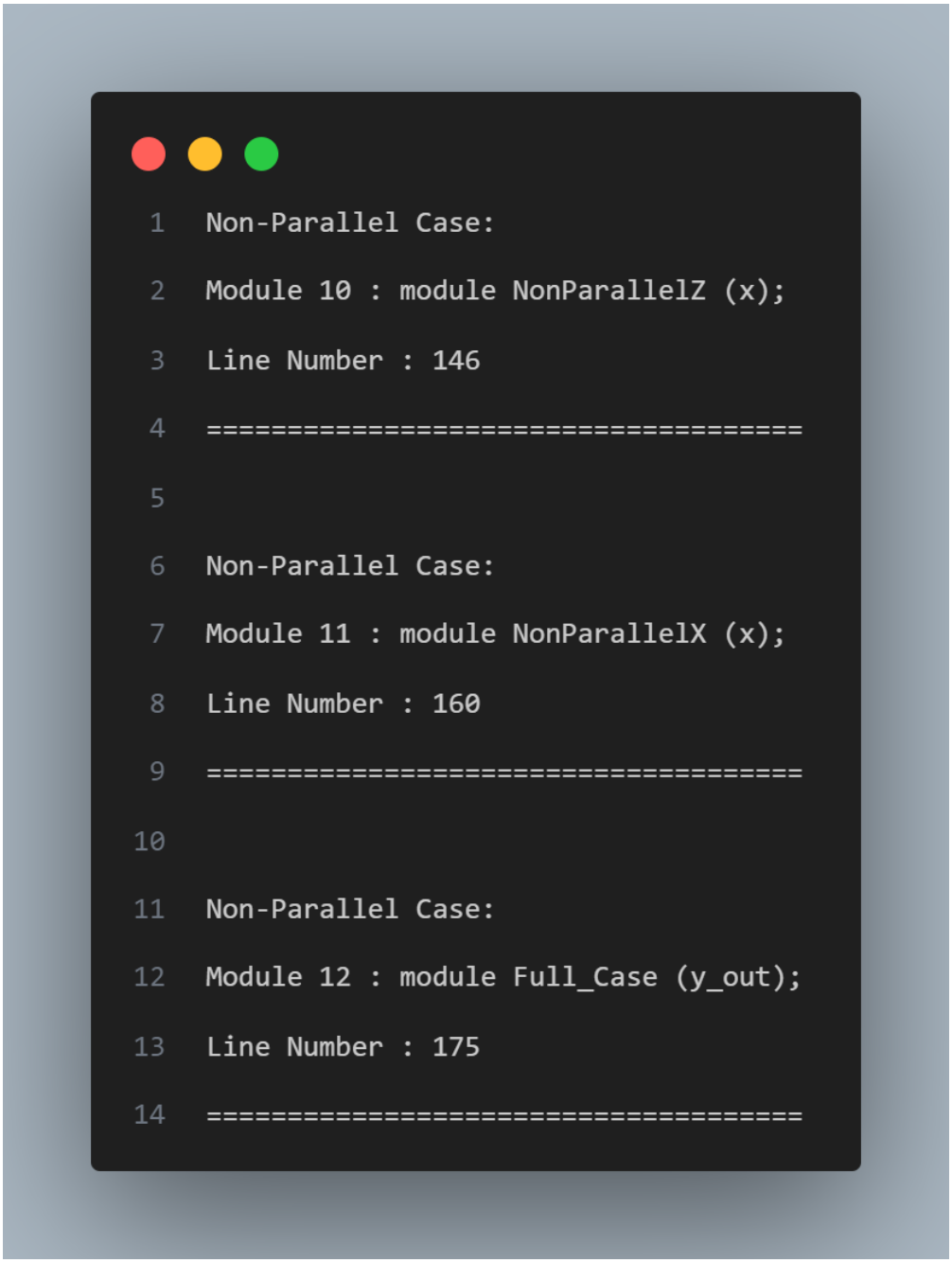


```
Module 13: module MultipleDrivers(input [1:0] x, output out);
Always Block: ['always', 'begin', 'y = y + 1;', 'end']
Line Number : 196
Always Block: ['always', 'begin', "y = 1'b0;", 'end']
Line Number : 200
Multidriven Variables: {'y'}
====================

Module 13: module MultipleDrivers(input [1:0] x, output out);
Line Number : 191
Assign Statements: [["assign out = 0'b1;", 'out']]
Multidriven Variables: {'out'}
====================
Module 14: Variable 'out2' is multidriven.
Assign Statements: [["assign out2 = 0'b1;", 'out2']]
Line Number : 208
Multidriven Variables: {'out2'}
====================================
```

5.6 Arithmetic Overflow

```
Possible Arithmetic Overflow in module 13 : module MultipleDrivers(input [1:0] x, output out);
Line Number : 196
Line: y = y + 1;
Left side size: 1
Right side size: 1
Operation List: [[['y'], ['y', '+', '1']]]
====================

Possible Arithmetic Overflow in module 15 : module ArithmeticOverflow(a,b,result);
Line Number : 222
Line: assign result = a + b;
Left side size: 4
Right side size: 4
Operation List: [[['assign', 'result'], ['a', '+', 'b']]]
====================

Possible Arithmetic Overflow in module 16 : module CombinationalFeedbackLoop(a, b);
Line Number : 231
Line: b = b + a;
Left side size: 1
Right side size: 1
Operation List: [[['b'], ['b', '+', 'a']]]
====================
```

5.7 Non Parallel Case



6 Limitations

6.1 Unreachable Block

- Cannot Work on Case Blocks (Works on if-else blocks)

6.2 Non-Full/Parallel Case

- Didn't two consecutive cases in the same module having same input reg

6.3 Inferring Latches

- Handled Most cases except when having a case that is a full case & doesn't have a default

6.4 Uninitialized Registers

- Handled all available cases

6.5 Arithmetic Overflow

- Handled all available cases

6.6 Multiple Drivers

- Handled all available cases