Faculty of engineering
Ain shams University

Verilog Lint

EDA Project Report CSE312

Presented to:

DR.Eman El Mandouh

ENG. Abdelrahman Sherif

Presented by:

Moaz Mohamed 22p0307
Seif elhusseiny 22p0215
Omar Walid 22p0166
Ibrahim Amr 19p2223
Osama Lasheen 19p6937
Ahmed Mohamed Alamin 22p0137
Ali Khaled Elemam 22p0260

Table of contents:

Introduction

Designing digital systems using Verilog can be both exciting and challenging. Verilog gives us the power to describe hardware behavior and structure, but even small mistakes in code can lead to errors that are difficult to catch during simulation or testing. Missing a sensitivity signal, using an uninitialized register, or accidentally creating an inferred latch can result in unexpected hardware behavior or costly design flaws. These issues can be frustrating, especially when working on complex designs.

This is where the **Verilog Linter** comes in. Think of it as a helpful assistant that carefully reviews your Verilog code to catch common mistakes before they become bigger problems. The linter analyzes your code, line by line, and checks for issues like:

- Arithmetic overflows (e.g., when a signal doesn't have enough bits to store a result).
- Undefined or uninitialized registers.
- Signals being driven by multiple sources.
- Problems in if or case statements, like missing conditions or incomplete logic.
- Misuse of blocking (=) and non-blocking (<=) assignments in different types of logic.

The goal of this project is to make the Verilog design process smoother and less error-prone by automating these checks. Instead of manually reviewing every line of code, the linter saves you time and effort. It generates an easy-to-read report that highlights potential issues, so you can focus on refining your design rather than hunting for bugs.

System Design and Architecture

The Verilog Linter is designed as a modular and extensible tool that analyzes Verilog code for common design issues. Its architecture is centered around simplicity, efficiency, and flexibility, allowing it to handle a wide range of Verilog constructs while being easy to enhance for future needs. Below is an overview of the system design and key components.

The Verilog Linter operates in the following stages:

1. **Input Parsing:**
   - The Verilog file is read and split into individual lines for processing.

   - Each line is analyzed using regular expressions to identify signal declarations, assignments, and blocks of code like always or case.
2. **Rule-Based Analysis:**
   - The linter applies a series of checks (rules) to detect potential issues such as undefined registers, arithmetic overflows, or inferred latches.

   - Each rule is implemented as a self-contained method, making it easy to add or modify rules.
3. **Error Tracking:**
   - Detected issues are categorized and stored in a centralized error log (self.errors) for reporting.

   - Each error entry includes the line number and a description of the issue.
4. **Report Generation:**
   - After all checks are completed, the linter generates a detailed report listing all detected violations.

   - The report highlights the type of error, the affected line, and a brief explanation.

Covered list of Violations:

. Arithmetic Overflow Check

. Multi-Driven Bus/Register

. Full/Parallel Case

. Duplicate Case check

. Infer latch

. Un-initialized Register

. Un-defined Register

. Sensitivity List Check

. Blocking/Non-Blocking check

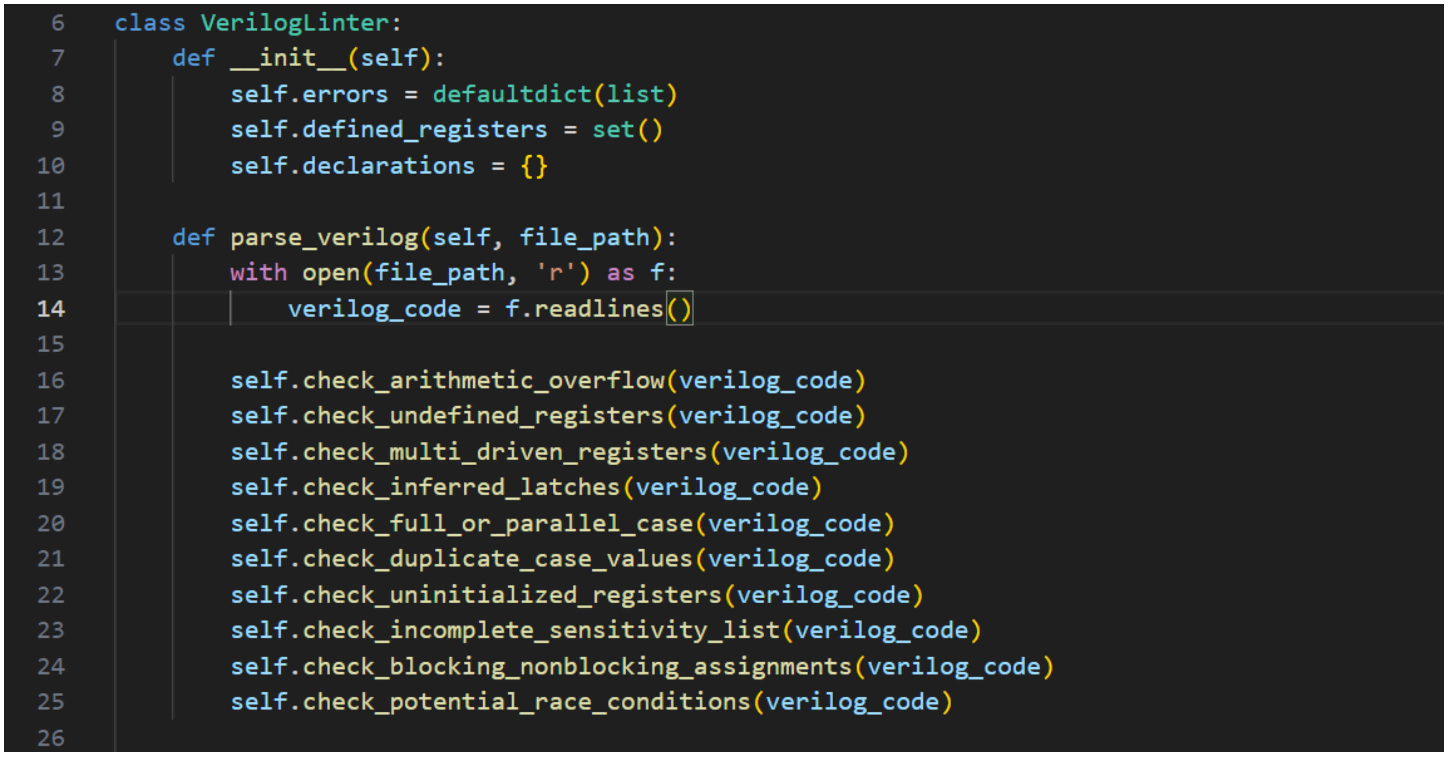
. Race conditions check

. Array out of bounds check

Lint project code:

Verilog linter class:

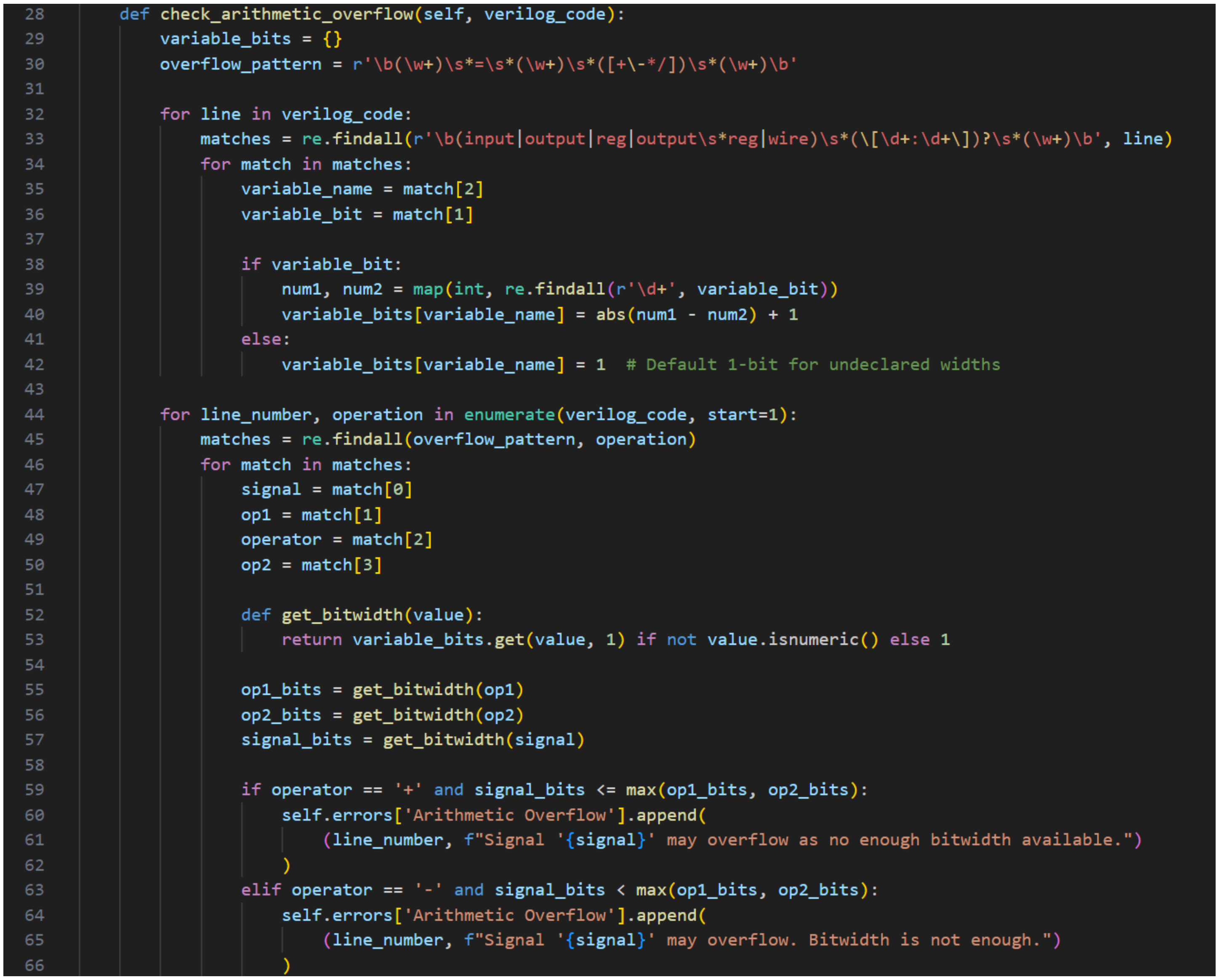
We made a class called Verilog linter and each method in it checks for a specific type of error and adds this error to the self.errors dictionary which will then be printed in the lint_report.txt file. We made a parse method that reads every line in the Verilog code and will be using the re (regular expression) library to get the part of the Verilog we want in each expression, and then we called all the check methods in this class



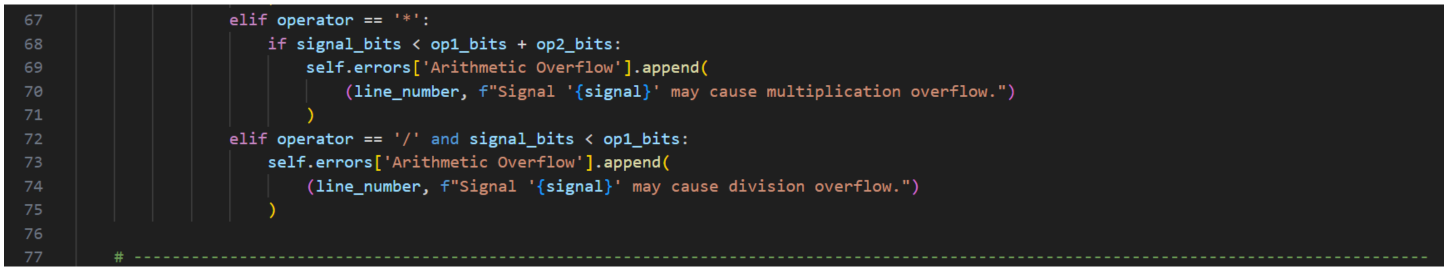
```python
class VerilogLinter:
    def __init__(self):
        self.errors = defaultdict(list)
        self.defined_registers = set()
        self.declarations = {}

    def parse_verilog(self, file_path):
        with open(file_path, 'r') as f:
            verilog_code = f.readlines()

        self.check_arithmetic_overflow(verilog_code)
        self.check_undefined_registers(verilog_code)
        self.check_multi_driven_registers(verilog_code)
        self.check_inferred_latches(verilog_code)
        self.check_full_or_parallel_case(verilog_code)
        self.check_duplicate_case_values(verilog_code)
        self.check_uninitialized_registers(verilog_code)
        self.check_incomplete_sensitivity_list(verilog_code)
        self.check_blocking_nonblocking_assignments(verilog_code)
        self.check_potential_race_conditions(verilog_code)

```

Arithmetic Overflow Check function:

We made a code for the Arithmetic Overflow that ensures the bit-width of a signal is sufficient to store the result of an arithmetic operation. It analyzes operations such as addition, subtraction, and multiplication by comparing operand and result bit-widths, flagging potential overflows. This check helps prevent unintended truncation or loss of data in hardware designs.



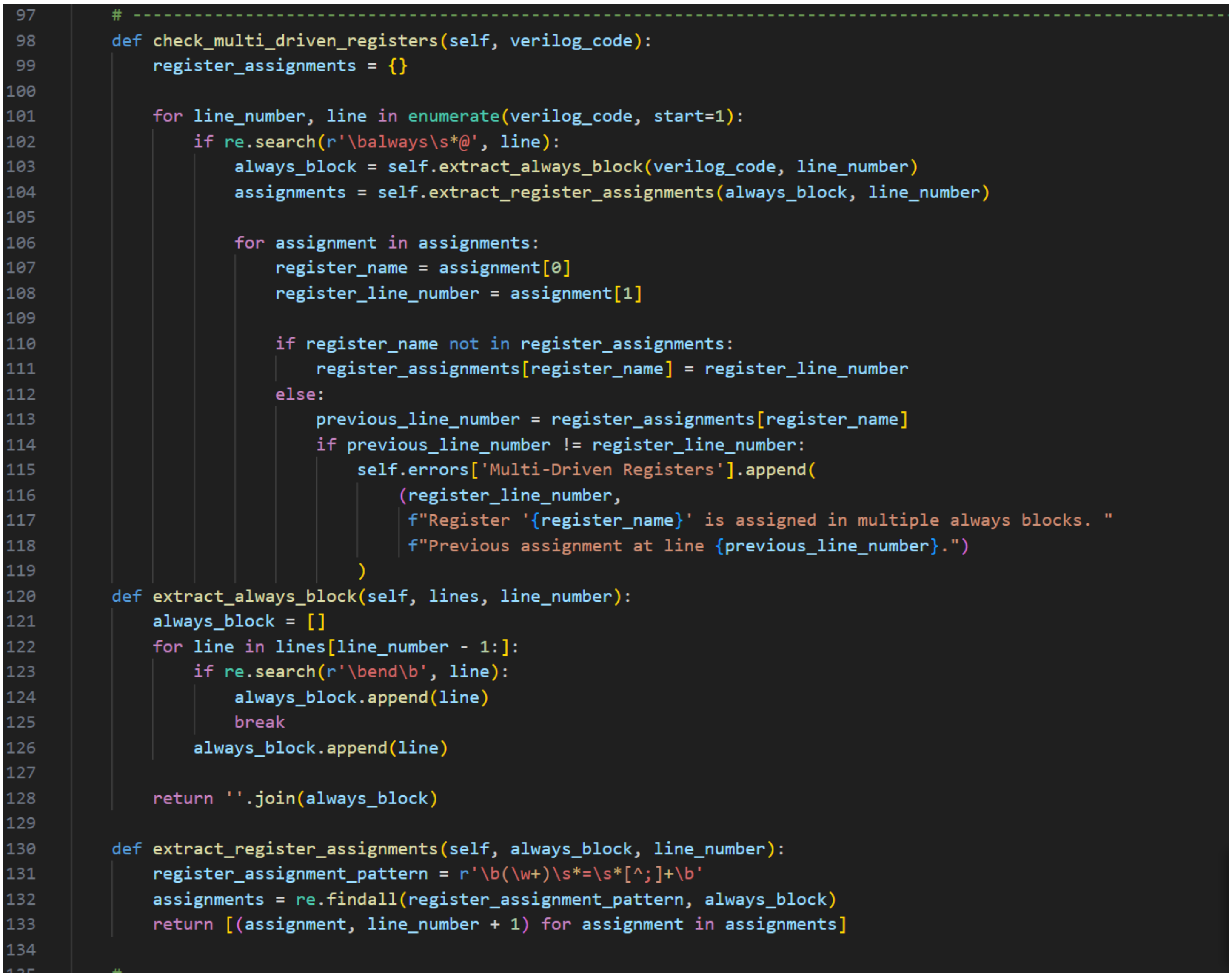
```python
    def check_arithmetic_overflow(self, verilog_code):
        variable_bits = {}
        overflow_pattern = r'\b(\w+)\s*=\s*(\w+)\s*([+\-*/])\s*(\w+)\b'

        for line in verilog_code:
            matches = re.findall(r'\b(input|output|reg|output\s*reg|wire)\s*(\[\d+:\d+\])?\s*(\w+)\b', line)
            for match in matches:
                variable_name = match[2]
                variable_bit = match[1]

                if variable_bit:
                    num1, num2 = map(int, re.findall(r'\d+', variable_bit))
                    variable_bits[variable_name] = abs(num1 - num2) + 1
                else:
                    variable_bits[variable_name] = 1  # Default 1-bit for undeclared widths

        for line_number, operation in enumerate(verilog_code, start=1):
            matches = re.findall(overflow_pattern, operation)
            for match in matches:
                signal = match[0]
                op1 = match[1]
                operator = match[2]
                op2 = match[3]

                def get_bitwidth(value):
                    return variable_bits.get(value, 1) if not value.isnumeric() else 1

                op1_bits = get_bitwidth(op1)
                op2_bits = get_bitwidth(op2)
                signal_bits = get_bitwidth(signal)

                if operator == '+' and signal_bits <= max(op1_bits, op2_bits):
                    self.errors['Arithmetic Overflow'].append(
                        (line_number, f"Signal '{signal}' may overflow as no enough bitwidth available.")
                    )
                elif operator == '-' and signal_bits < max(op1_bits, op2_bits):
                    self.errors['Arithmetic Overflow'].append(
                        (line_number, f"Signal '{signal}' may overflow. Bitwidth is not enough.")
                    )
```



```
                elif operator == '*':
                    if signal_bits < op1_bits + op2_bits:
                        self.errors['Arithmetic Overflow'].append(
                            (line_number, f"Signal '{signal}' may cause multiplication overflow.")
                        )
                elif operator == '/' and signal_bits < op1_bits:
                    self.errors['Arithmetic Overflow'].append(
                        (line_number, f"Signal '{signal}' may cause division overflow.")
                    )

    # -----------------------------------------------------------------------------------------------------------------------------
```

Multi-Driven Bus/Register function:

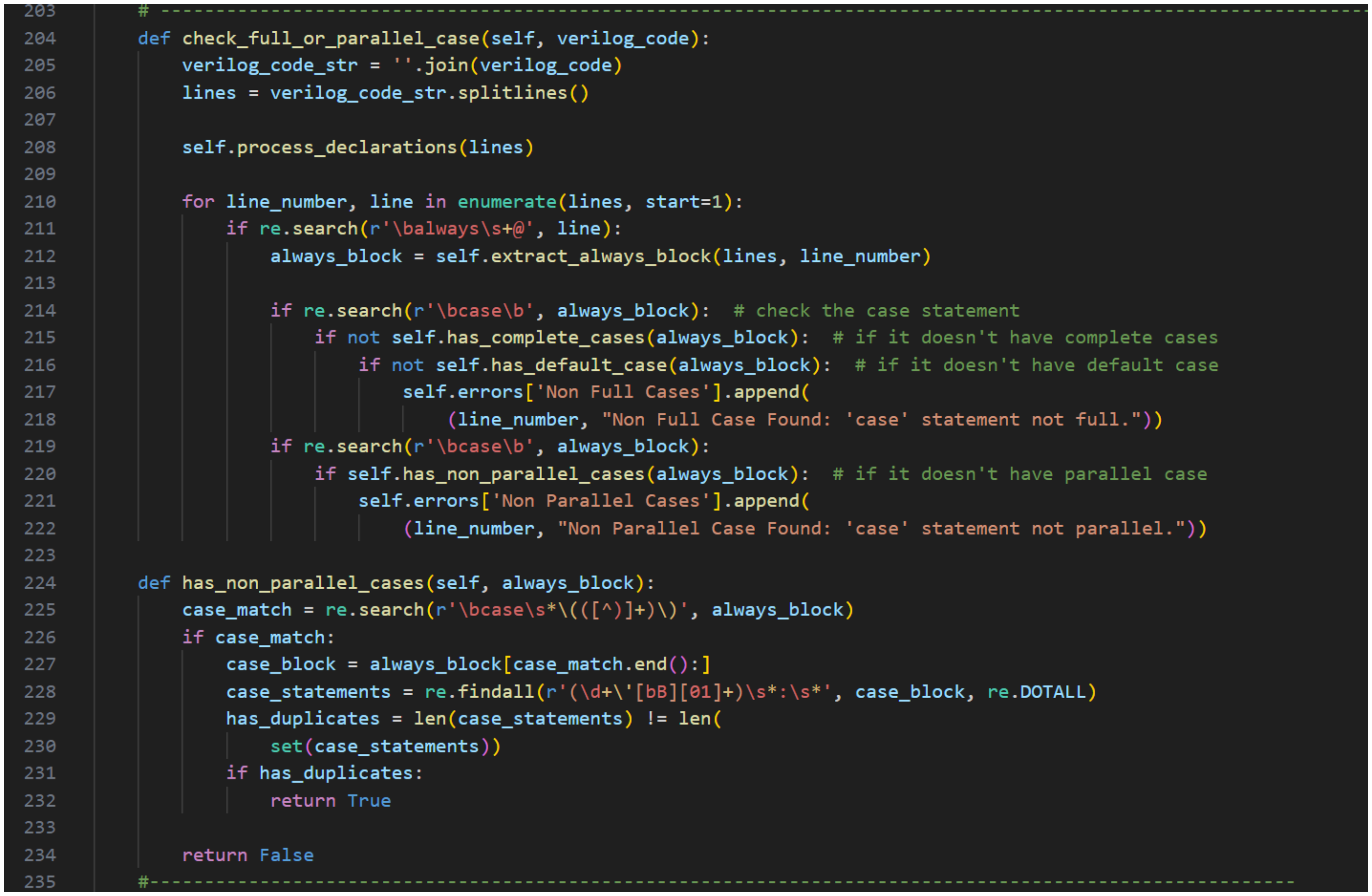
Multi Driven Overflow Check Function: In this function we made a pattern to detect always blocks then we made a pattern to detect any assignment operation (=) in this always block. For every assignment we find we put the register name and the line which this register was found in a dictionary. If we find that the register was already in the dictionary, and it has a different line number meaning that it was defined in another always block. We will add this violation in the self. Error and the lines which made this violation.



```python
# ------------------------------------------------------------------------------------------------------------------
def check_multi_driven_registers(self, verilog_code):
    register_assignments = {}

    for line_number, line in enumerate(verilog_code, start=1):
        if re.search(r'\balways\s*@', line):
            always_block = self.extract_always_block(verilog_code, line_number)
            assignments = self.extract_register_assignments(always_block, line_number)

            for assignment in assignments:
                register_name = assignment[0]
                register_line_number = assignment[1]

                if register_name not in register_assignments:
                    register_assignments[register_name] = register_line_number
                else:
                    previous_line_number = register_assignments[register_name]
                    if previous_line_number != register_line_number:
                        self.errors['Multi-Driven Registers'].append(
                            (register_line_number,
                             f"Register '{register_name}' is assigned in multiple always blocks. "
                             f"Previous assignment at line {previous_line_number}.")
                        )
def extract_always_block(self, lines, line_number):
    always_block = []
    for line in lines[line_number - 1:]:
        if re.search(r'\bend\b', line):
            always_block.append(line)
            break
        always_block.append(line)

    return ''.join(always_block)

def extract_register_assignments(self, always_block, line_number):
    register_assignment_pattern = r'\b(\w+)\s*=\s*[^;]+\b'
    assignments = re.findall(register_assignment_pattern, always_block)
    return [(assignment, line_number + 1) for assignment in assignments]

#
```

Full/Parallel Case function

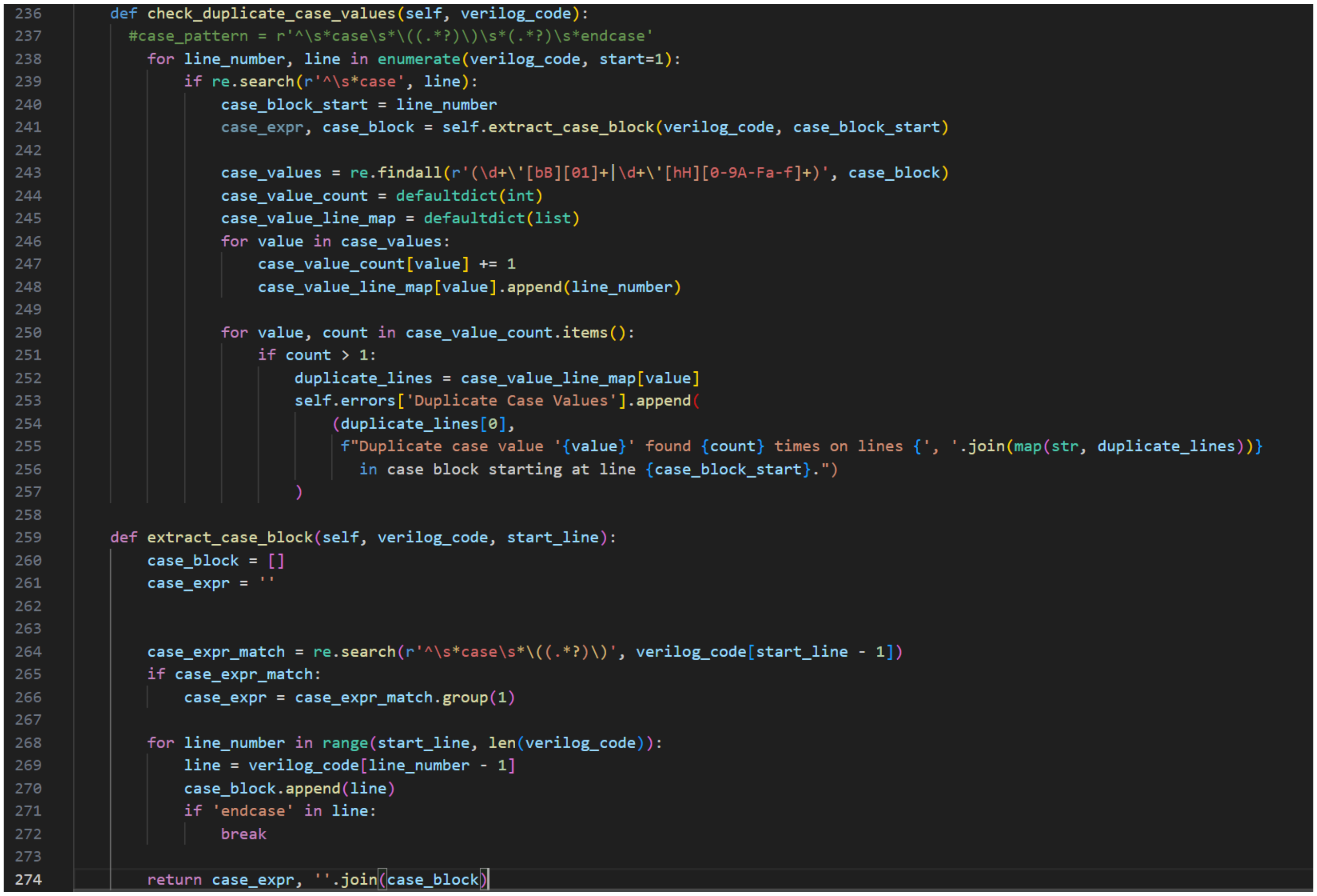
The check_full_or_parallel_case function analyzes Verilog code to ensure case statements in always blocks are both "full" (cover all possible cases or include a default case) and "parallel" (no duplicate conditions). It processes Verilog code to identify always blocks and extracts their contents. For each case statement found, it checks completeness using the has_complete_cases and has_default_case methods. It also checks for non-parallel cases by identifying duplicate case conditions with has_non_parallel_cases. Errors for incomplete or non-parallel cases are recorded with their line numbers and detailed messages.



```
    # ------------------------------------------------------------------------------------------------------
    def check_full_or_parallel_case(self, verilog_code):
        verilog_code_str = ''.join(verilog_code)
        lines = verilog_code_str.splitlines()

        self.process_declarations(lines)

        for line_number, line in enumerate(lines, start=1):
            if re.search(r'\balways\s+@', line):
                always_block = self.extract_always_block(lines, line_number)

                if re.search(r'\bcase\b', always_block):  # check the case statement
                    if not self.has_complete_cases(always_block):  # if it doesn't have complete cases
                        if not self.has_default_case(always_block):  # if it doesn't have default case
                            self.errors['Non Full Cases'].append(
                                (line_number, "Non Full Case Found: 'case' statement not full."))
                if re.search(r'\bcase\b', always_block):
                    if self.has_non_parallel_cases(always_block):  # if it doesn't have parallel case
                        self.errors['Non Parallel Cases'].append(
                            (line_number, "Non Parallel Case Found: 'case' statement not parallel."))

    def has_non_parallel_cases(self, always_block):
        case_match = re.search(r'\bcase\s*\(([^)]+)\)', always_block)
        if case_match:
            case_block = always_block[case_match.end():]
            case_statements = re.findall(r'(\d+\'[bB][01]+)\s*:\s*', case_block, re.DOTALL)
            has_duplicates = len(case_statements) != len(
                set(case_statements))
            if has_duplicates:
                return True

        return False
    #-----------------------------------------------------------------------------------------------------
```

<u>Duplicate Case function:</u>

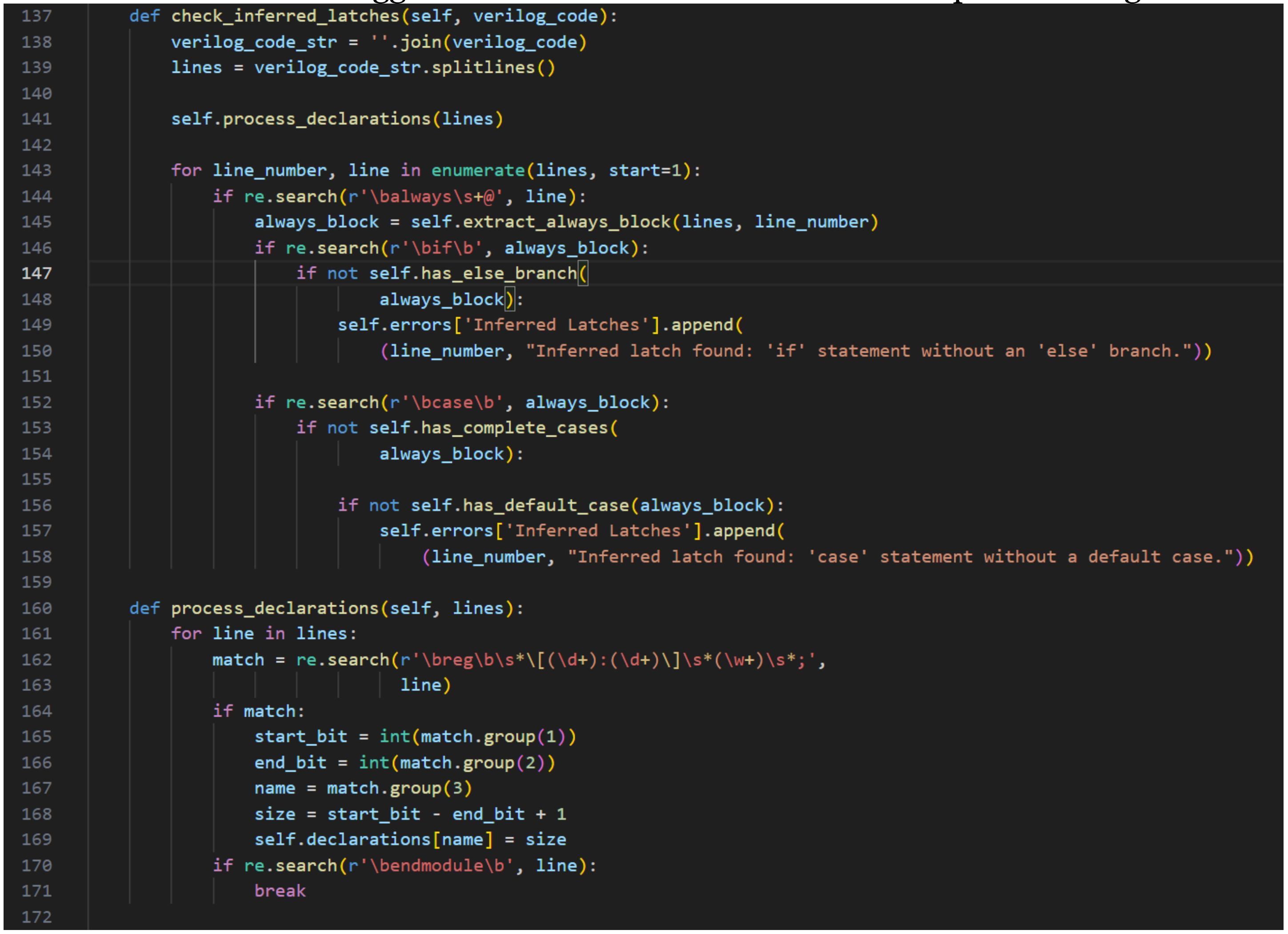
The check_duplicate_case_values function analyzes Verilog case blocks to identify duplicate values assigned to multiple cases. It identifies case blocks by detecting lines starting with case and extracting the block content using the extract_case_block helper function, which collects all lines from the starting case to the corresponding endcase. Within each block, it locates all case values (e.g., binary, hexadecimal) and tracks their occurrences and line numbers. If any value appears more than once, it logs an error in self.errors under "Duplicate Case Values," including the duplicate value, the number of occurrences, the lines where it appears, and the starting line of the case block.



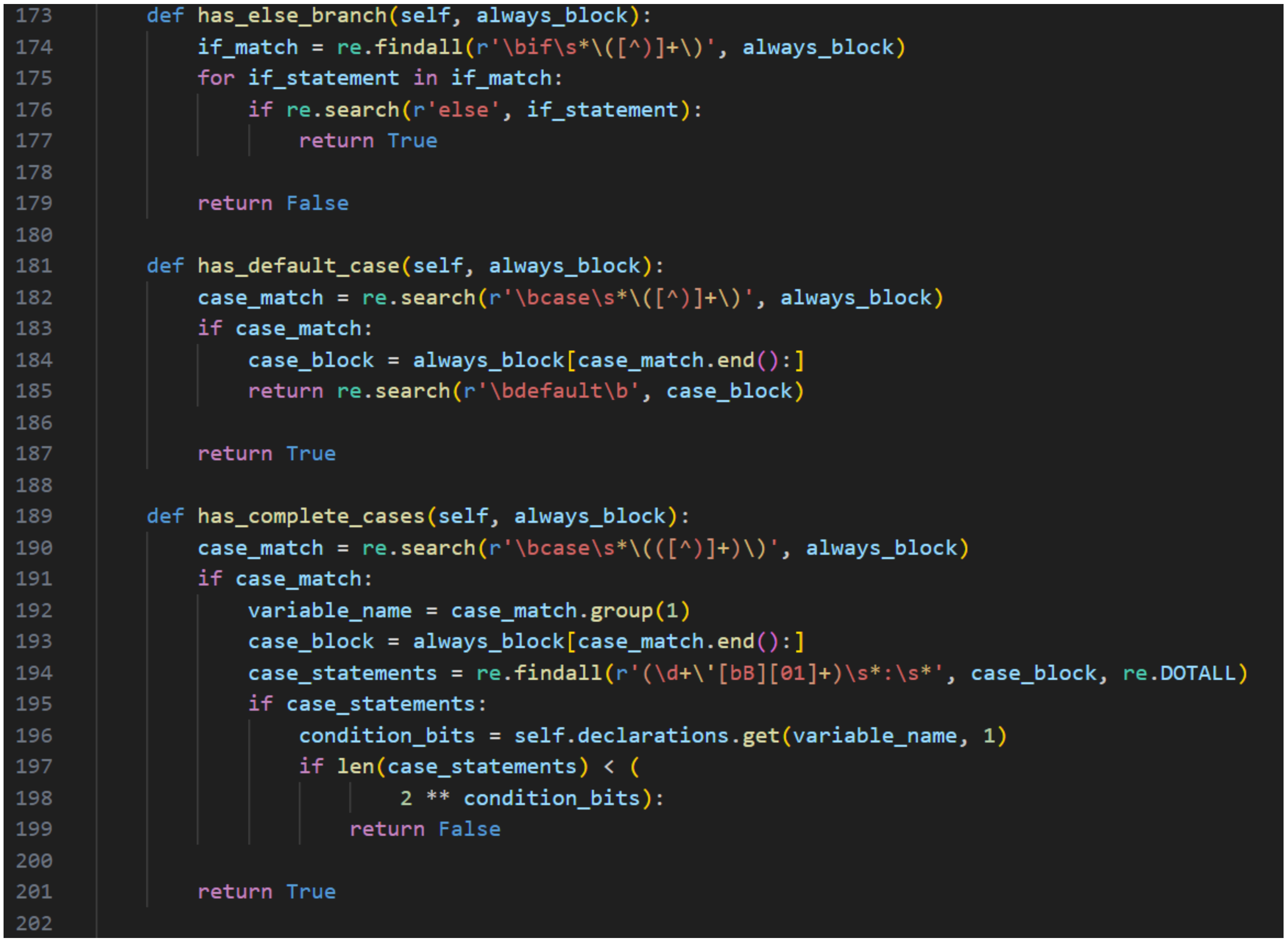
```python
    def check_duplicate_case_values(self, verilog_code):
      #case_pattern = r'^\s*case\s*\((.*?)\)\s*(.*?)\s*endcase'
        for line_number, line in enumerate(verilog_code, start=1):
            if re.search(r'^\s*case', line):
                case_block_start = line_number
                case_expr, case_block = self.extract_case_block(verilog_code, case_block_start)

                case_values = re.findall(r'(\d+\'[bB][01]+|\d+\'[hH][0-9A-Fa-f]+)', case_block)
                case_value_count = defaultdict(int)
                case_value_line_map = defaultdict(list)
                for value in case_values:
                    case_value_count[value] += 1
                    case_value_line_map[value].append(line_number)

                for value, count in case_value_count.items():
                    if count > 1:
                        duplicate_lines = case_value_line_map[value]
                        self.errors['Duplicate Case Values'].append(
                            (duplicate_lines[0],
                             f"Duplicate case value '{value}' found {count} times on lines {', '.join(map(str, duplicate_lines))}
                               in case block starting at line {case_block_start}.")
                        )

    def extract_case_block(self, verilog_code, start_line):
        case_block = []
        case_expr = ''


        case_expr_match = re.search(r'^\s*case\s*\((.*?)\)', verilog_code[start_line - 1])
        if case_expr_match:
            case_expr = case_expr_match.group(1)

        for line_number in range(start_line, len(verilog_code)):
            line = verilog_code[line_number - 1]
            case_block.append(line)
            if 'endcase' in line:
                break

        return case_expr, ''.join(case_block)
```

Infer latch check function:

The check_inferred_latches function inspects Verilog code to identify situations where inferred latches might be created, which can occur when conditional constructs are incomplete. It examines always blocks and flags if statements without else branches and case statements that lack a default case or fail to cover all possible conditions. Supporting functions help process declarations (process_declarations extracts signal sizes from reg definitions) and verify completeness (has_complete_cases checks if a case covers all possible values, and has_default_case checks for a default case). Errors for inferred latches are logged with their line numbers and descriptive messages.



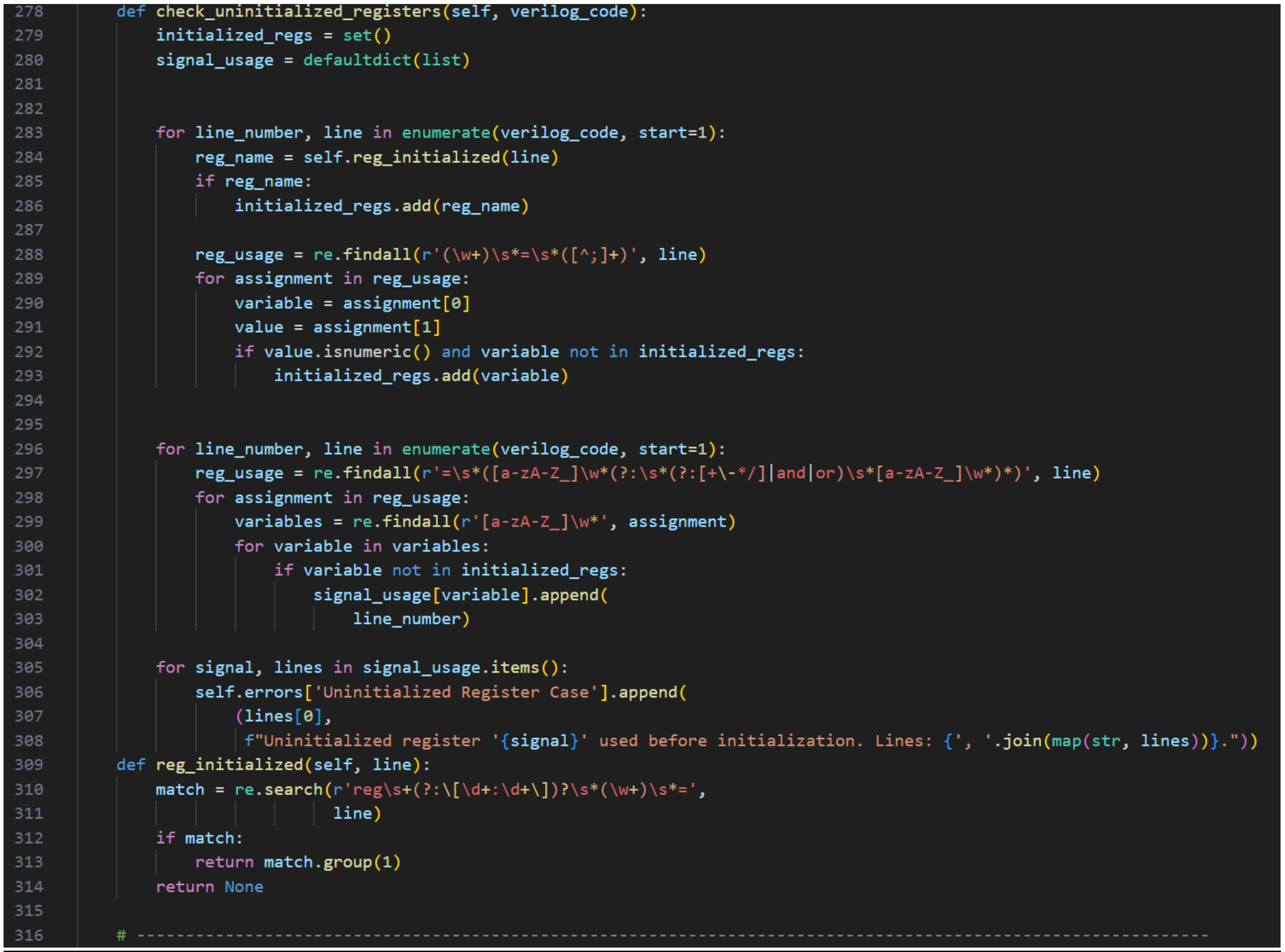
```
    def check_inferred_latches(self, verilog_code):
        verilog_code_str = ''.join(verilog_code)
        lines = verilog_code_str.splitlines()

        self.process_declarations(lines)

        for line_number, line in enumerate(lines, start=1):
            if re.search(r'\balways\s+@', line):
                always_block = self.extract_always_block(lines, line_number)
                if re.search(r'\bif\b', always_block):
                    if not self.has_else_branch(
                            always_block):
                        self.errors['Inferred Latches'].append(
                            (line_number, "Inferred latch found: 'if' statement without an 'else' branch."))

                if re.search(r'\bcase\b', always_block):
                    if not self.has_complete_cases(
                            always_block):

                        if not self.has_default_case(always_block):
                            self.errors['Inferred Latches'].append(
                                (line_number, "Inferred latch found: 'case' statement without a default case."))

    def process_declarations(self, lines):
        for line in lines:
            match = re.search(r'\breg\b\s*\[(\d+):(\d+)\]\s*(\w+)\s*;',
                              line)
            if match:
                start_bit = int(match.group(1))
                end_bit = int(match.group(2))
                name = match.group(3)
                size = start_bit - end_bit + 1
                self.declarations[name] = size
            if re.search(r'\bendmodule\b', line):
                break
```



```python
    def has_else_branch(self, always_block):
        if_match = re.findall(r'\bif\s*\([^)]+\)', always_block)
        for if_statement in if_match:
            if re.search(r'else', if_statement):
                return True

        return False

    def has_default_case(self, always_block):
        case_match = re.search(r'\bcase\s*\([^)]+\)', always_block)
        if case_match:
            case_block = always_block[case_match.end():]
            return re.search(r'\bdefault\b', case_block)

        return True

    def has_complete_cases(self, always_block):
        case_match = re.search(r'\bcase\s*\(([^)]+)\)', always_block)
        if case_match:
            variable_name = case_match.group(1)
            case_block = always_block[case_match.end():]
            case_statements = re.findall(r'(\d+\'[bB][01]+)\s*:\s*', case_block, re.DOTALL)
            if case_statements:
                condition_bits = self.declarations.get(variable_name, 1)
                if len(case_statements) < (
                        2 ** condition_bits):
                    return False

        return True

```

Un-initialized Register check function:

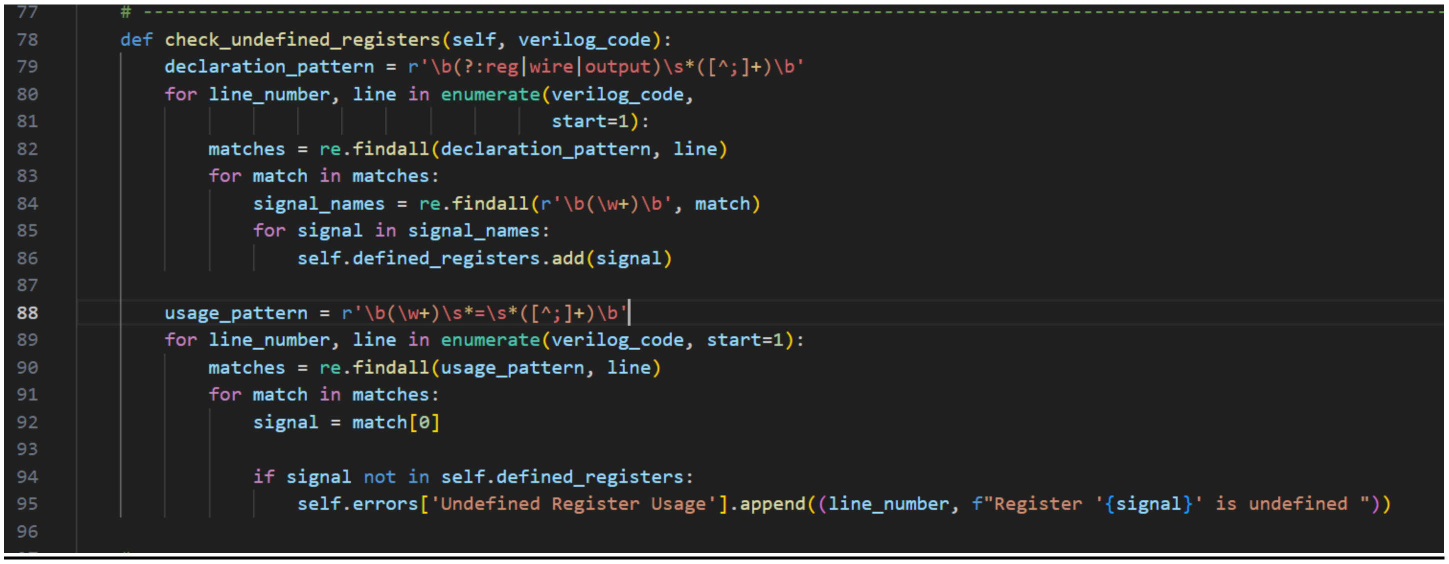
The check_uninitialized_registers function analyzes Verilog code to detect instances where registers are used before being initialized. It maintains a set of initialized registers and tracks signal usage across lines of code. Registers explicitly initialized during declaration or assigned numeric values are added to the initialized_regs set. For other assignments, it identifies variables used in expressions and checks if they are uninitialized. Any uninitialized register usage is logged in self.errors with the line number and a message indicating the issue. The helper function reg_initialized detects registers initialized during their declaration.



```python
    def check_uninitialized_registers(self, verilog_code):
        initialized_regs = set()
        signal_usage = defaultdict(list)


        for line_number, line in enumerate(verilog_code, start=1):
            reg_name = self.reg_initialized(line)
            if reg_name:
                initialized_regs.add(reg_name)

            reg_usage = re.findall(r'(\w+)\s*=\s*([^;]+)', line)
            for assignment in reg_usage:
                variable = assignment[0]
                value = assignment[1]
                if value.isnumeric() and variable not in initialized_regs:
                    initialized_regs.add(variable)


        for line_number, line in enumerate(verilog_code, start=1):
            reg_usage = re.findall(r'=\s*([a-zA-Z_]\w*(?:\s*(?:[+\-*/]|and|or)\s*[a-zA-Z_]\w*)*)', line)
            for assignment in reg_usage:
                variables = re.findall(r'[a-zA-Z_]\w*', assignment)
                for variable in variables:
                    if variable not in initialized_regs:
                        signal_usage[variable].append(
                            line_number)

        for signal, lines in signal_usage.items():
            self.errors['Uninitialized Register Case'].append(
                (lines[0],
                 f"Uninitialized register '{signal}' used before initialization. Lines: {', '.join(map(str, lines))}."))
    def reg_initialized(self, line):
        match = re.search(r'reg\s+(?:\[\d+:\d+\])?\s*(\w+)\s*=',
                          line)
        if match:
            return match.group(1)
        return None

    # ----------------------------------------------------------------------------------------------------------
```

Un-defined Register function:

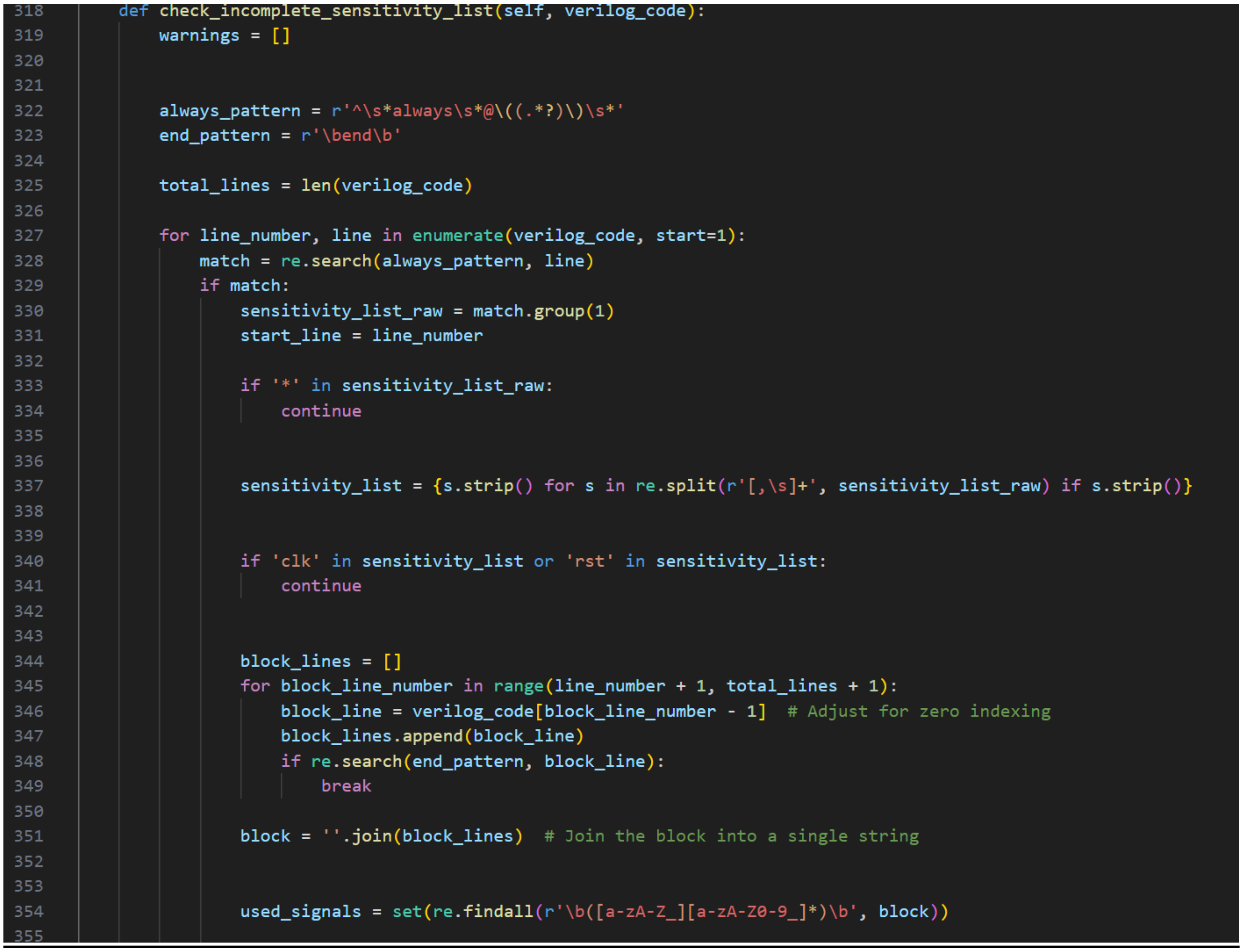
The check_undefined_registers function analyzes Verilog code to detect instances where registers are used without being defined. It first identifies defined signals (reg, wire, or output) by parsing the code with a regular expression and stores these in a set. Next, it checks assignment statements to ensure that signals being assigned values are already defined. If an undefined signal is found, an error is recorded with its line number and a descriptive message. The function systematically uses regular expressions to extract and validate signal names, updating an errors dictionary with any issues related to undefined register usage.



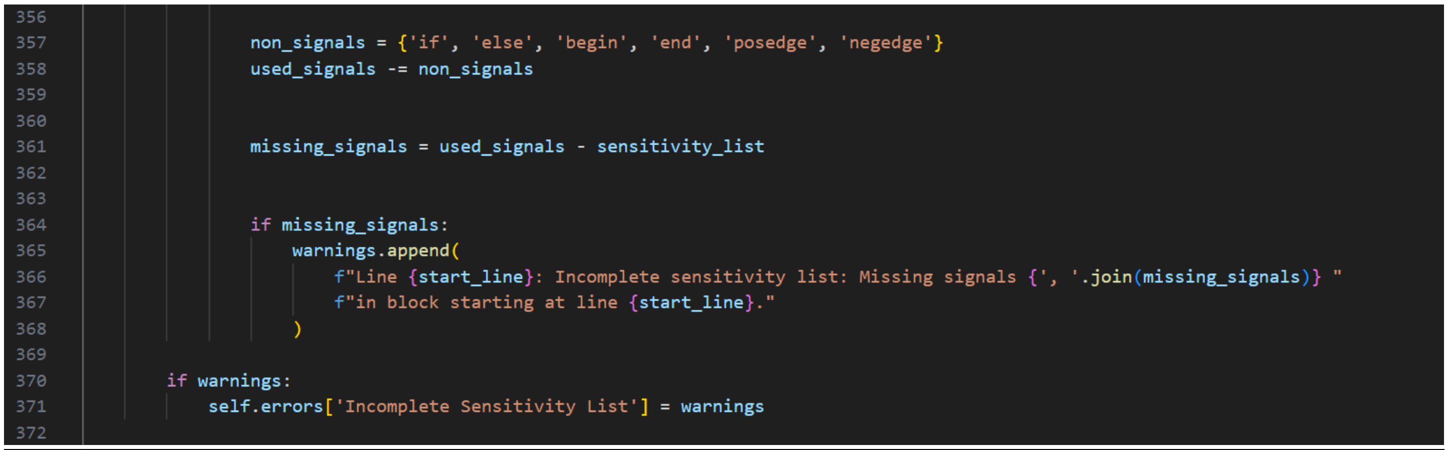
```python
    # ------------------------------------------------------------------------------------------------------------------------
    def check_undefined_registers(self, verilog_code):
        declaration_pattern = r'\b(?:reg|wire|output)\s*([^;]+)\b'
        for line_number, line in enumerate(verilog_code,
                                           start=1):
            matches = re.findall(declaration_pattern, line)
            for match in matches:
                signal_names = re.findall(r'\b(\w+)\b', match)
                for signal in signal_names:
                    self.defined_registers.add(signal)

        usage_pattern = r'\b(\w+)\s*=\s*([^;]+)\b'
        for line_number, line in enumerate(verilog_code, start=1):
            matches = re.findall(usage_pattern, line)
            for match in matches:
                signal = match[0]

                if signal not in self.defined_registers:
                    self.errors['Undefined Register Usage'].append((line_number, f"Register '{signal}' is undefined "))

```

Sensitivity List Check Function:

The check_incomplete_sensitivity_list function inspects Verilog always blocks to identify missing signals in their sensitivity lists. It processes each always block by extracting the sensitivity list and verifying whether it includes all signals used in the block. Blocks with wildcard (*) sensitivity lists or those containing clk or rst are excluded from checks. For other blocks, it collects all signals used within the block, filters out non-signals (e.g., if, else, begin), and compares them against the sensitivity list. If any signals are missing, a warning is recorded in self.errors with the block's starting line number and the missing signals.



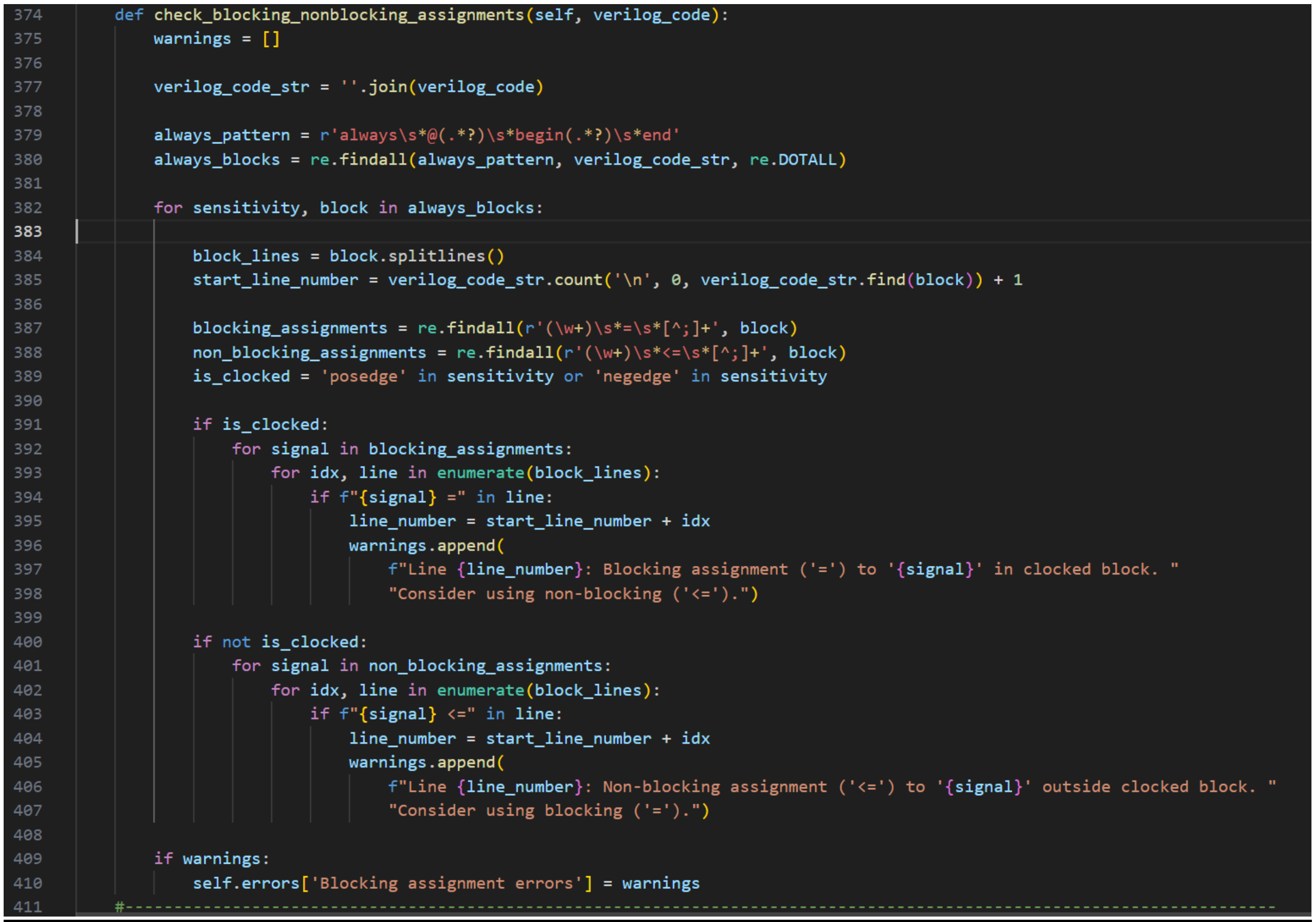
```python
    def check_incomplete_sensitivity_list(self, verilog_code):
        warnings = []


        always_pattern = r'^\s*always\s*@\((.*?)\)\s*'
        end_pattern = r'\bend\b'

        total_lines = len(verilog_code)

        for line_number, line in enumerate(verilog_code, start=1):
            match = re.search(always_pattern, line)
            if match:
                sensitivity_list_raw = match.group(1)
                start_line = line_number

                if '*' in sensitivity_list_raw:
                    continue


                sensitivity_list = {s.strip() for s in re.split(r'[,\s]+', sensitivity_list_raw) if s.strip()}


                if 'clk' in sensitivity_list or 'rst' in sensitivity_list:
                    continue


                block_lines = []
                for block_line_number in range(line_number + 1, total_lines + 1):
                    block_line = verilog_code[block_line_number - 1]  # Adjust for zero indexing
                    block_lines.append(block_line)
                    if re.search(end_pattern, block_line):
                        break

                block = ''.join(block_lines)  # Join the block into a single string


                used_signals = set(re.findall(r'\b([a-zA-Z_][a-zA-Z0-9_]*)\b', block))
```



```python

                non_signals = {'if', 'else', 'begin', 'end', 'posedge', 'negedge'}
                used_signals -= non_signals


                missing_signals = used_signals - sensitivity_list


                if missing_signals:
                    warnings.append(
                        f"Line {start_line}: Incomplete sensitivity list: Missing signals {', '.join(missing_signals)} "
                        f"in block starting at line {start_line}."
                    )

        if warnings:
            self.errors['Incomplete Sensitivity List'] = warnings

```

Blocking/Non-Blocking check function:

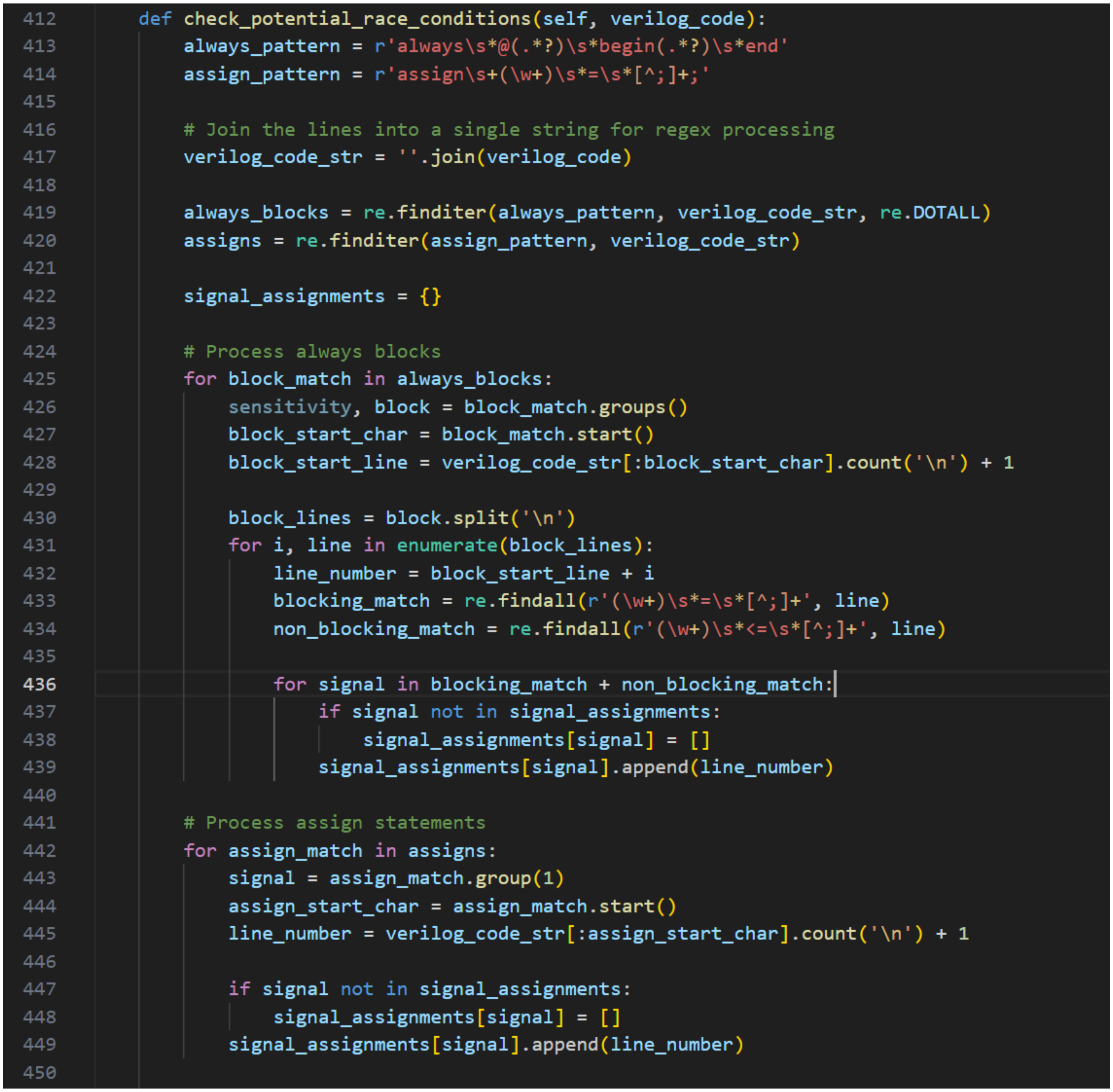
The check_blocking_nonblocking_assignments function inspects Verilog code to identify improper use of blocking (=) and non-blocking (<=) assignments within always blocks. It analyzes each always block to determine if it is clocked (contains posedge or negedge in its sensitivity list). In clocked blocks, blocking assignments are flagged as improper, as non-blocking assignments should be used. Conversely, in non-clocked blocks, non-blocking assignments are flagged because blocking assignments are more appropriate. The function logs warnings in self.errors with details about the line number, the assignment type, and a recommendation to use the correct assignment operator.



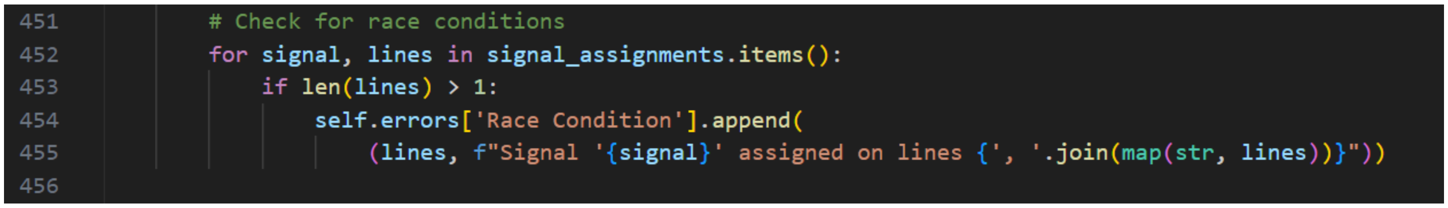
```python
    def check_blocking_nonblocking_assignments(self, verilog_code):
        warnings = []

        verilog_code_str = ''.join(verilog_code)

        always_pattern = r'always\s*@(.*?)\s*begin(.*?)\s*end'
        always_blocks = re.findall(always_pattern, verilog_code_str, re.DOTALL)

        for sensitivity, block in always_blocks:

            block_lines = block.splitlines()
            start_line_number = verilog_code_str.count('\n', 0, verilog_code_str.find(block)) + 1

            blocking_assignments = re.findall(r'(\w+)\s*=\s*[^;]+', block)
            non_blocking_assignments = re.findall(r'(\w+)\s*<=\s*[^;]+', block)
            is_clocked = 'posedge' in sensitivity or 'negedge' in sensitivity

            if is_clocked:
                for signal in blocking_assignments:
                    for idx, line in enumerate(block_lines):
                        if f"{signal} =" in line:
                            line_number = start_line_number + idx
                            warnings.append(
                                f"Line {line_number}: Blocking assignment ('=') to '{signal}' in clocked block. "
                                "Consider using non-blocking ('<=').")

            if not is_clocked:
                for signal in non_blocking_assignments:
                    for idx, line in enumerate(block_lines):
                        if f"{signal} <=" in line:
                            line_number = start_line_number + idx
                            warnings.append(
                                f"Line {line_number}: Non-blocking assignment ('<=') to '{signal}' outside clocked block. "
                                "Consider using blocking ('=').")

        if warnings:
            self.errors['Blocking assignment errors'] = warnings
    #---------------------------------------------------------------------------------------------------------------------
```

Race conditions check function:

The `check_potential_race_conditions` function analyzes Verilog code to detect potential race conditions caused by multiple assignments to the same signal. It processes `always` blocks and `assign` statements, extracting the signals being assigned values and recording the line numbers of their occurrences. For `always` blocks, it identifies both blocking (`=`) and non-blocking (`<=`) assignments, while for `assign` statements, it captures continuous assignments. If any signal is assigned on multiple lines, it is flagged as a potential race condition. The details, including the signal name and the conflicting line numbers, are recorded in the `self.errors` under "Race Condition."



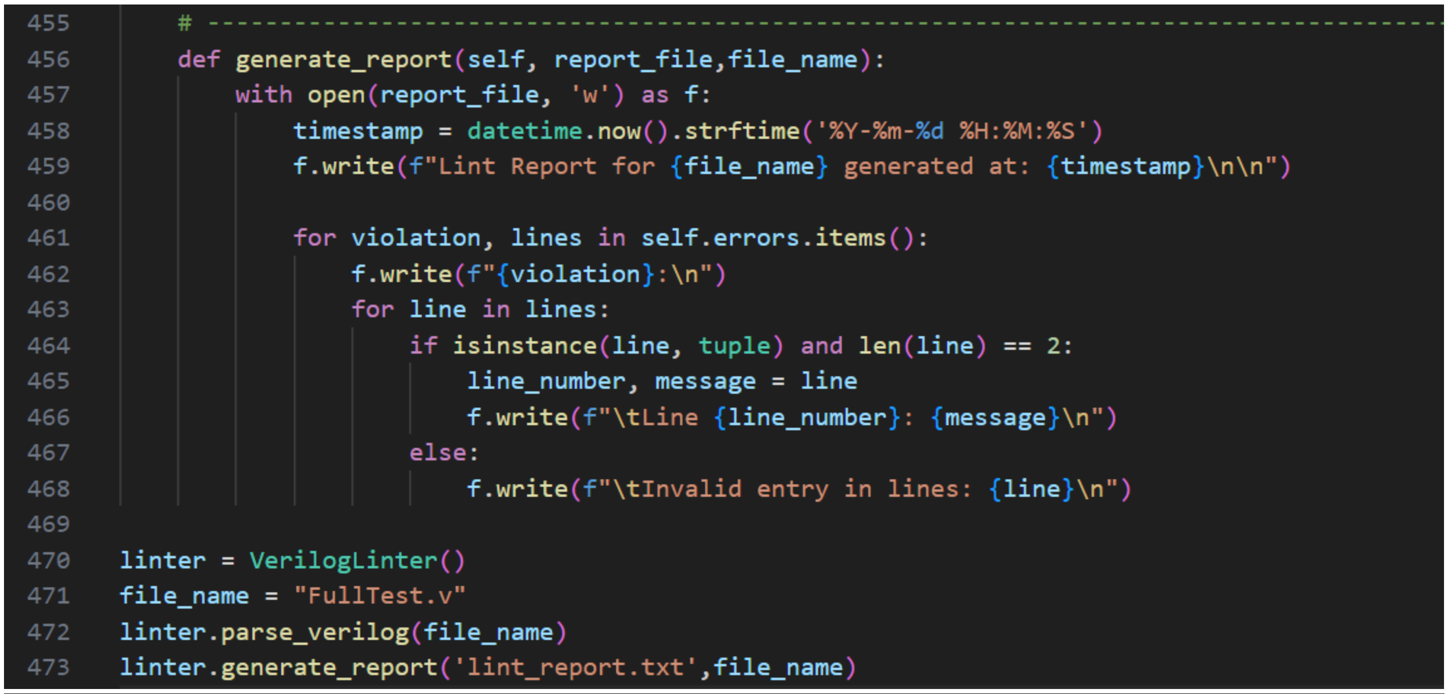
```python
    def check_potential_race_conditions(self, verilog_code):
        always_pattern = r'always\s*@(.*?)\s*begin(.*?)\s*end'
        assign_pattern = r'assign\s+(\w+)\s*=\s*[^;]+;'

        # Join the lines into a single string for regex processing
        verilog_code_str = ''.join(verilog_code)

        always_blocks = re.finditer(always_pattern, verilog_code_str, re.DOTALL)
        assigns = re.finditer(assign_pattern, verilog_code_str)

        signal_assignments = {}

        # Process always blocks
        for block_match in always_blocks:
            sensitivity, block = block_match.groups()
            block_start_char = block_match.start()
            block_start_line = verilog_code_str[:block_start_char].count('\n') + 1

            block_lines = block.split('\n')
            for i, line in enumerate(block_lines):
                line_number = block_start_line + i
                blocking_match = re.findall(r'(\w+)\s*=\s*[^;]+', line)
                non_blocking_match = re.findall(r'(\w+)\s*<=\s*[^;]+', line)

                for signal in blocking_match + non_blocking_match:
                    if signal not in signal_assignments:
                        signal_assignments[signal] = []
                    signal_assignments[signal].append(line_number)

        # Process assign statements
        for assign_match in assigns:
            signal = assign_match.group(1)
            assign_start_char = assign_match.start()
            line_number = verilog_code_str[:assign_start_char].count('\n') + 1

            if signal not in signal_assignments:
                signal_assignments[signal] = []
            signal_assignments[signal].append(line_number)

```



```python
        # Check for race conditions
        for signal, lines in signal_assignments.items():
            if len(lines) > 1:
                self.errors['Race Condition'].append(
                    (lines, f"Signal '{signal}' assigned on lines {', '.join(map(str, lines))}"))

```

Generate Report Function:

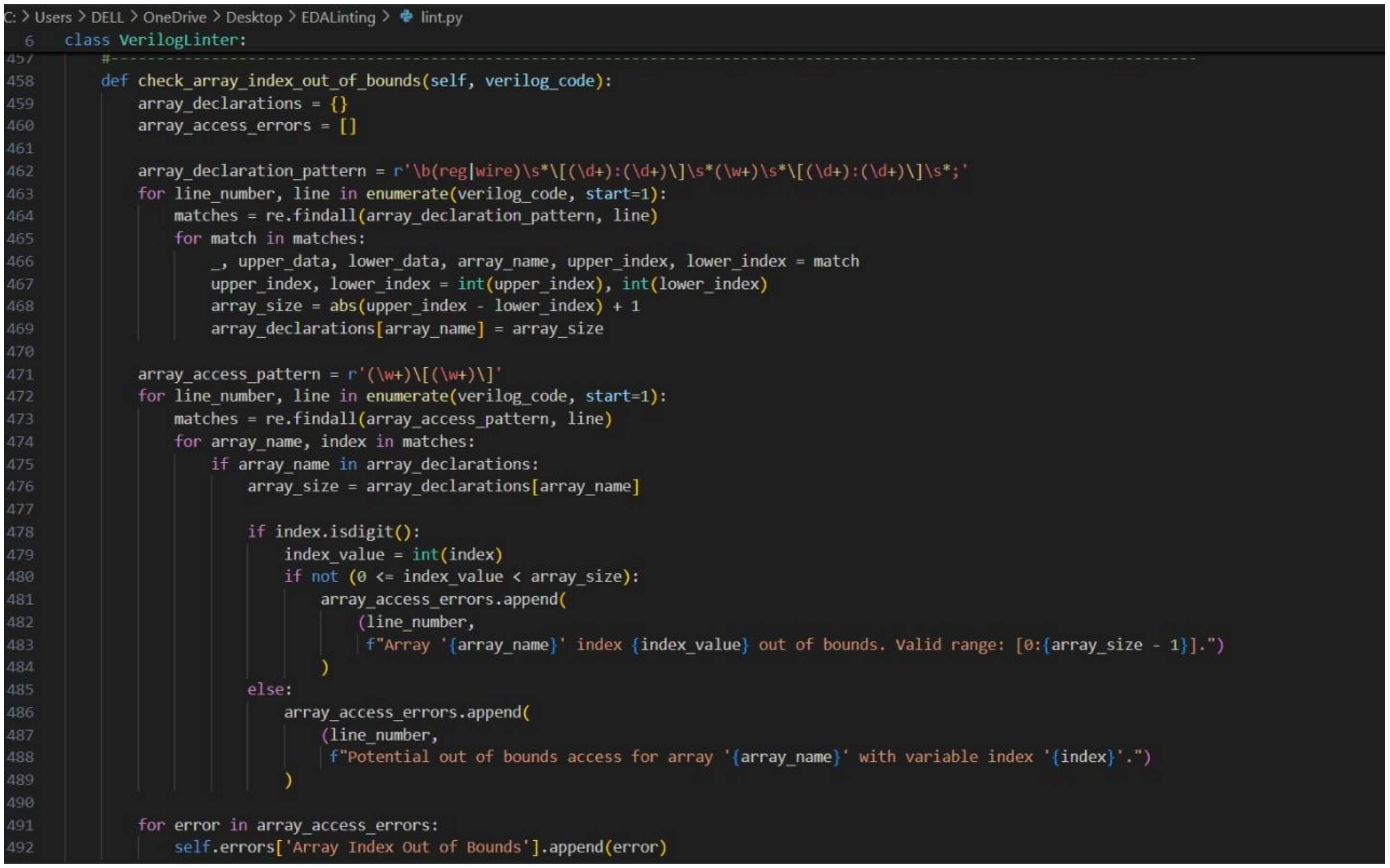
The generate_report function creates a linting report summarizing all detected code violations. It writes the report to the specified report_file, including a timestamp and the analyzed file name. For each type of violation stored in self.errors, it lists the violation name and details each instance, specifying the line number and error message. If an error entry is invalid (e.g., not a tuple with two elements), it logs it as an invalid entry. This ensures a comprehensive and structured overview of code issues.



```python
    # --------------------------------------------------------------------------------
    def generate_report(self, report_file,file_name):
        with open(report_file, 'w') as f:
            timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
            f.write(f"Lint Report for {file_name} generated at: {timestamp}\n\n")

            for violation, lines in self.errors.items():
                f.write(f"{violation}:\n")
                for line in lines:
                    if isinstance(line, tuple) and len(line) == 2:
                        line_number, message = line
                        f.write(f"\tLine {line_number}: {message}\n")
                    else:
                        f.write(f"\tInvalid entry in lines: {line}\n")

linter = VerilogLinter()
file_name = "FullTest.v"
linter.parse_verilog(file_name)
linter.generate_report('lint_report.txt',file_name)
```

Array out of bounds check function:

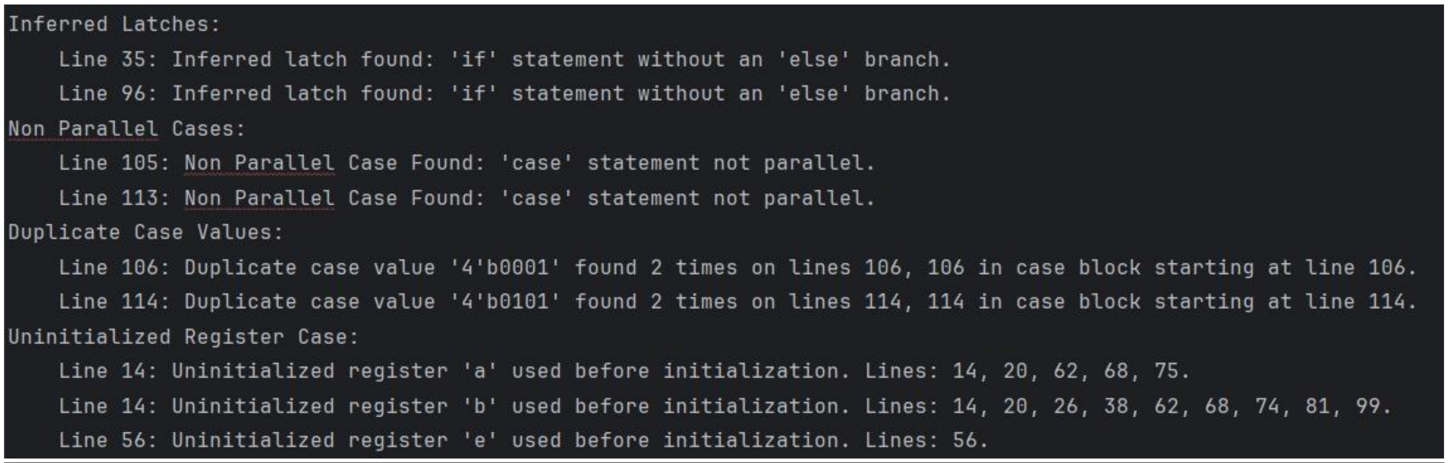
The check_array_index_out_of_bounds function in the VerilogLinter class analyzes Verilog code to detect array index out-of-bound errors. It uses regex patterns to identify array declarations (reg or wire with bounds) and accesses (array_name[index]). Array sizes are calculated from declared bounds and stored in the array_declarations dictionary. For each array access, if the index is numeric, it ensures the index is within [0, array_size - 1]; otherwise, variable indices are flagged as potential out-of-bounds issues. Errors, including line numbers and details, are collected in array_access_errors and appended to self.errors['Array Index Out of Bounds']. While the code effectively handles numeric and variable indices with clear error reporting, it lacks support for multi-dimensional arrays and variable range inference, which could enhance its robustness.



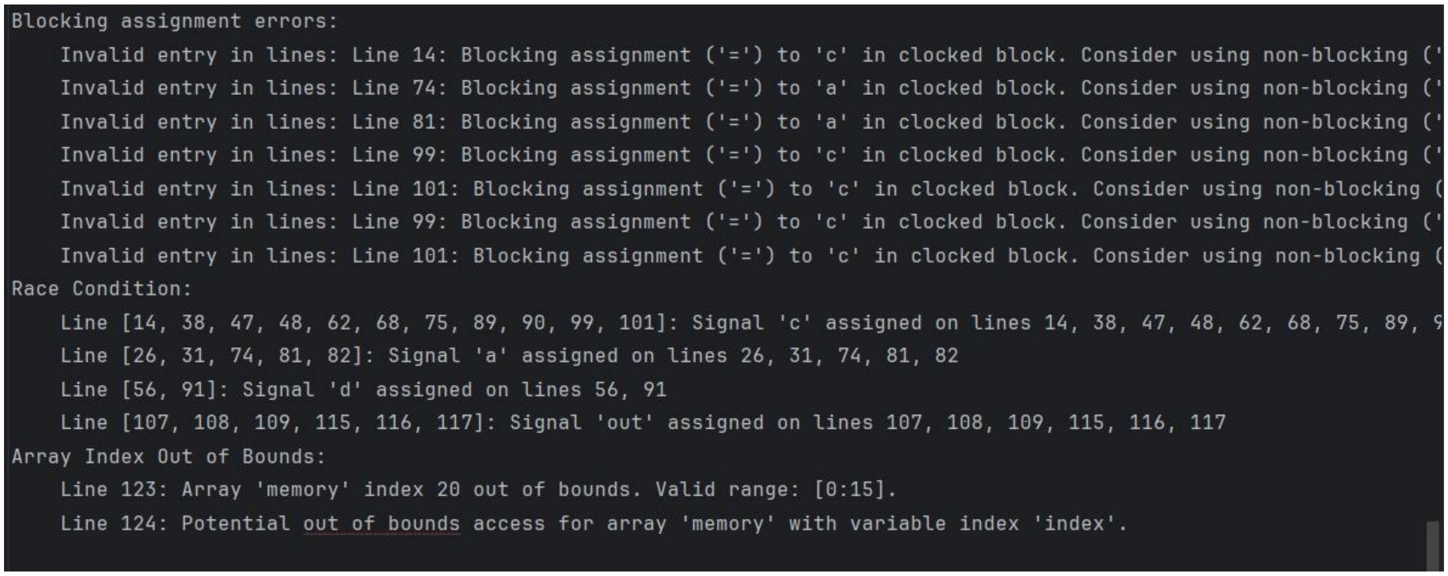
```
C: > Users > DELL > OneDrive > Desktop > EDALinting > lint.py
class VerilogLinter:
    #-----------------------------------------------------------------------------------------------------------------------
    def check_array_index_out_of_bounds(self, verilog_code):
        array_declarations = {}
        array_access_errors = []

        array_declaration_pattern = r'\b(reg|wire)\s*\[(\d+):(\d+)\]\s*(\w+)\s*\[(\d+):(\d+)\]\s*;'
        for line_number, line in enumerate(verilog_code, start=1):
            matches = re.findall(array_declaration_pattern, line)
            for match in matches:
                _, upper_data, lower_data, array_name, upper_index, lower_index = match
                upper_index, lower_index = int(upper_index), int(lower_index)
                array_size = abs(upper_index - lower_index) + 1
                array_declarations[array_name] = array_size

        array_access_pattern = r'(\w+)\[(\w+)\]'
        for line_number, line in enumerate(verilog_code, start=1):
            matches = re.findall(array_access_pattern, line)
            for array_name, index in matches:
                if array_name in array_declarations:
                    array_size = array_declarations[array_name]

                    if index.isdigit():
                        index_value = int(index)
                        if not (0 <= index_value < array_size):
                            array_access_errors.append(
                                (line_number,
                                 f"Array '{array_name}' index {index_value} out of bounds. Valid range: [0:{array_size - 1}].")
                            )
                    else:
                        array_access_errors.append(
                            (line_number,
                             f"Potential out of bounds access for array '{array_name}' with variable index '{index}'.")
                        )

        for error in array_access_errors:
            self.errors['Array Index Out of Bounds'].append(error)
```

Sample Outputs:

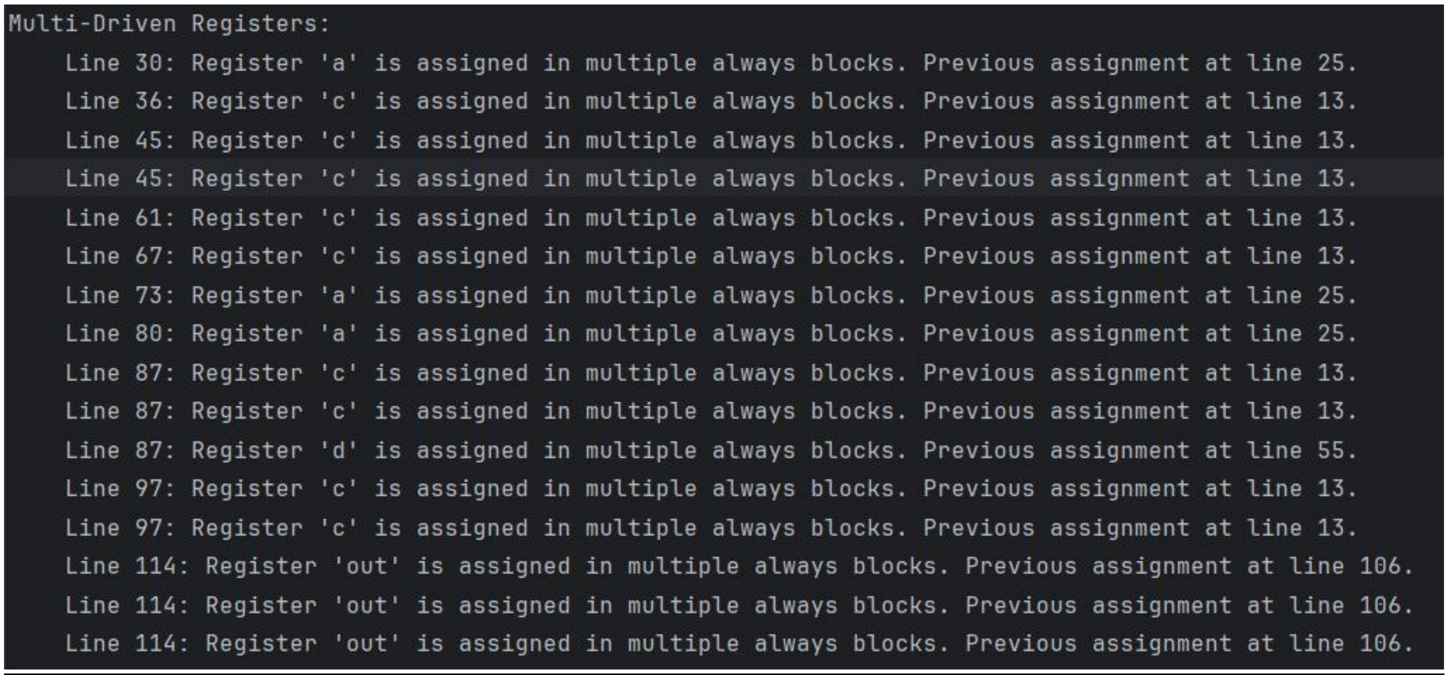
Here are some sample outputs for the linter



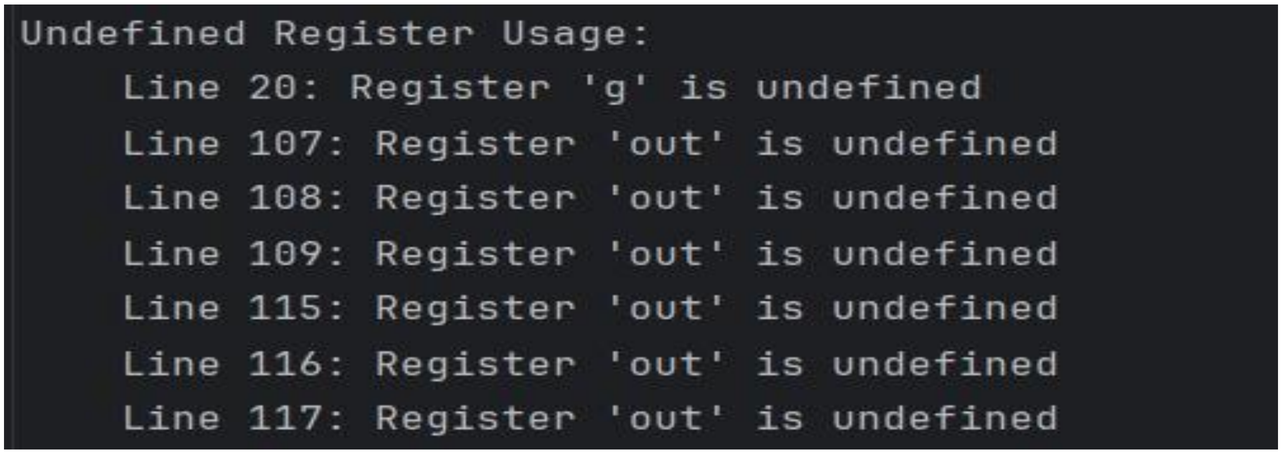
```
Inferred Latches:
    Line 35: Inferred latch found: 'if' statement without an 'else' branch.
    Line 96: Inferred latch found: 'if' statement without an 'else' branch.
Non Parallel Cases:
    Line 105: Non Parallel Case Found: 'case' statement not parallel.
    Line 113: Non Parallel Case Found: 'case' statement not parallel.
Duplicate Case Values:
    Line 106: Duplicate case value '4'b0001' found 2 times on lines 106, 106 in case block starting at line 106.
    Line 114: Duplicate case value '4'b0101' found 2 times on lines 114, 114 in case block starting at line 114.
Uninitialized Register Case:
    Line 14: Uninitialized register 'a' used before initialization. Lines: 14, 20, 62, 68, 75.
    Line 14: Uninitialized register 'b' used before initialization. Lines: 14, 20, 26, 38, 62, 68, 74, 81, 99.
    Line 56: Uninitialized register 'e' used before initialization. Lines: 56.
```



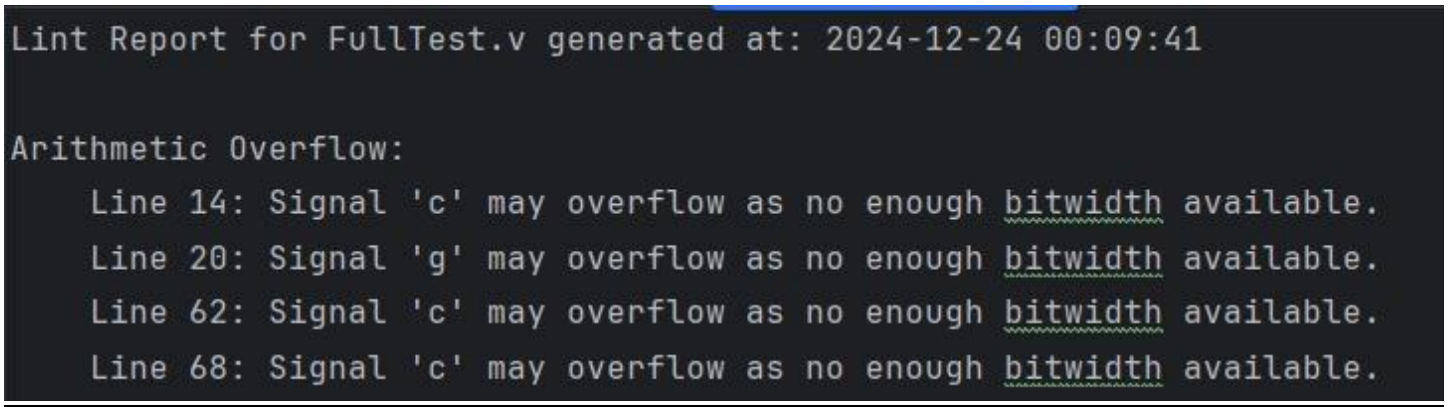
```
Blocking assignment errors:
    Invalid entry in lines: Line 14: Blocking assignment ('=') to 'c' in clocked block. Consider using non-blocking ('
    Invalid entry in lines: Line 74: Blocking assignment ('=') to 'a' in clocked block. Consider using non-blocking ('
    Invalid entry in lines: Line 81: Blocking assignment ('=') to 'a' in clocked block. Consider using non-blocking ('
    Invalid entry in lines: Line 99: Blocking assignment ('=') to 'c' in clocked block. Consider using non-blocking ('
    Invalid entry in lines: Line 101: Blocking assignment ('=') to 'c' in clocked block. Consider using non-blocking (
    Invalid entry in lines: Line 99: Blocking assignment ('=') to 'c' in clocked block. Consider using non-blocking ('
    Invalid entry in lines: Line 101: Blocking assignment ('=') to 'c' in clocked block. Consider using non-blocking (
Race Condition:
    Line [14, 38, 47, 48, 62, 68, 75, 89, 90, 99, 101]: Signal 'c' assigned on lines 14, 38, 47, 48, 62, 68, 75, 89, 9
    Line [26, 31, 74, 81, 82]: Signal 'a' assigned on lines 26, 31, 74, 81, 82
    Line [56, 91]: Signal 'd' assigned on lines 56, 91
    Line [107, 108, 109, 115, 116, 117]: Signal 'out' assigned on lines 107, 108, 109, 115, 116, 117
Array Index Out of Bounds:
    Line 123: Array 'memory' index 20 out of bounds. Valid range: [0:15].
    Line 124: Potential out of bounds access for array 'memory' with variable index 'index'.
```



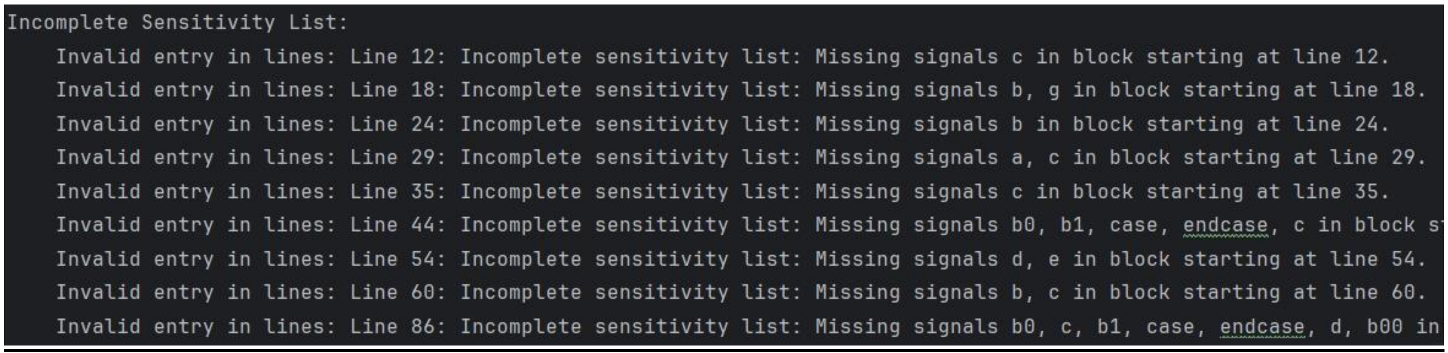
```
Multi-Driven Registers:
    Line 30: Register 'a' is assigned in multiple always blocks. Previous assignment at line 25.
    Line 36: Register 'c' is assigned in multiple always blocks. Previous assignment at line 13.
    Line 45: Register 'c' is assigned in multiple always blocks. Previous assignment at line 13.
    Line 45: Register 'c' is assigned in multiple always blocks. Previous assignment at line 13.
    Line 61: Register 'c' is assigned in multiple always blocks. Previous assignment at line 13.
    Line 67: Register 'c' is assigned in multiple always blocks. Previous assignment at line 13.
    Line 73: Register 'a' is assigned in multiple always blocks. Previous assignment at line 25.
    Line 80: Register 'a' is assigned in multiple always blocks. Previous assignment at line 25.
    Line 87: Register 'c' is assigned in multiple always blocks. Previous assignment at line 13.
    Line 87: Register 'c' is assigned in multiple always blocks. Previous assignment at line 13.
    Line 87: Register 'd' is assigned in multiple always blocks. Previous assignment at line 55.
    Line 97: Register 'c' is assigned in multiple always blocks. Previous assignment at line 13.
    Line 97: Register 'c' is assigned in multiple always blocks. Previous assignment at line 13.
    Line 114: Register 'out' is assigned in multiple always blocks. Previous assignment at line 106.
    Line 114: Register 'out' is assigned in multiple always blocks. Previous assignment at line 106.
    Line 114: Register 'out' is assigned in multiple always blocks. Previous assignment at line 106.
```



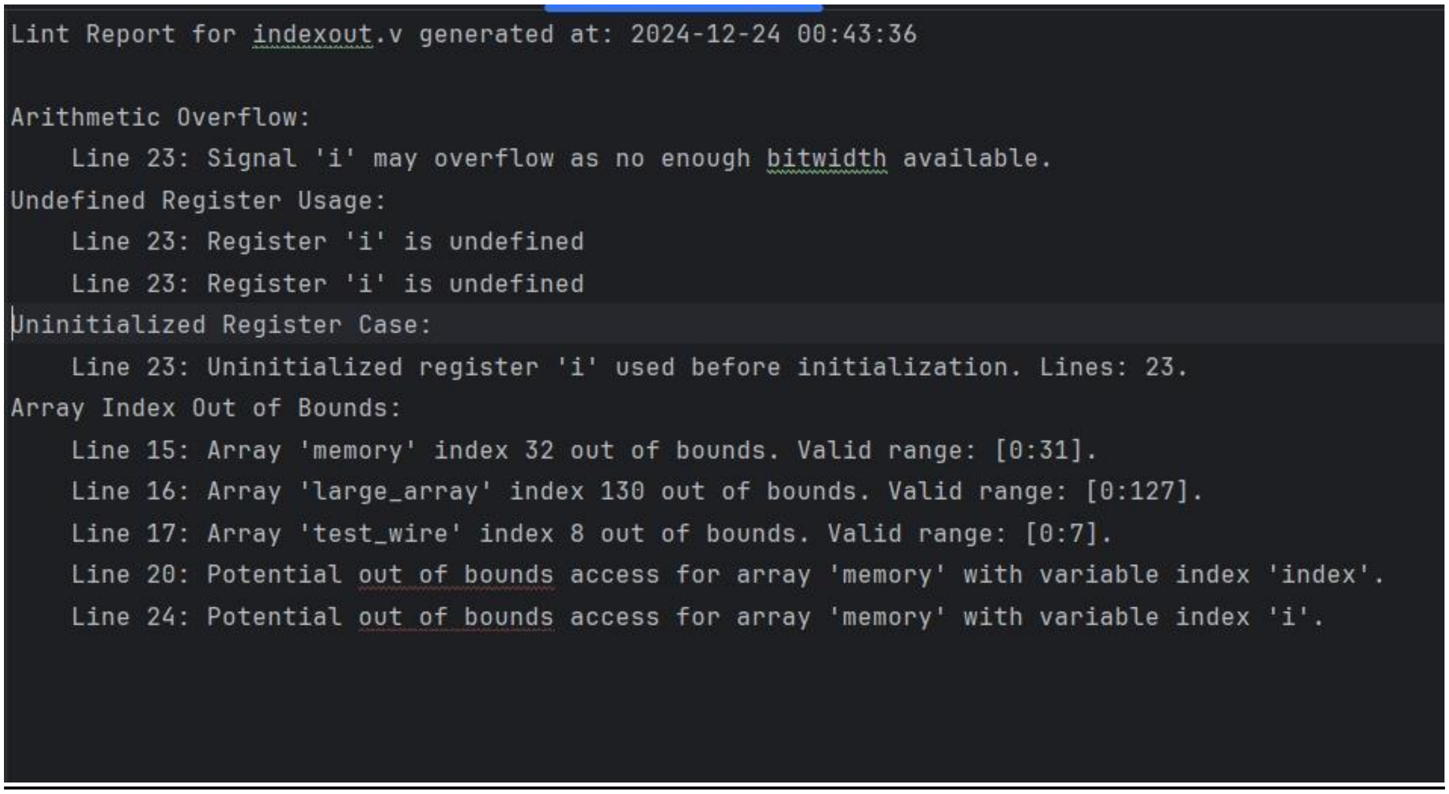
```
Undefined Register Usage:
    Line 20: Register 'g' is undefined
    Line 107: Register 'out' is undefined
    Line 108: Register 'out' is undefined
    Line 109: Register 'out' is undefined
    Line 115: Register 'out' is undefined
    Line 116: Register 'out' is undefined
    Line 117: Register 'out' is undefined
```



```
Lint Report for FullTest.v generated at: 2024-12-24 00:09:41

Arithmetic Overflow:
    Line 14: Signal 'c' may overflow as no enough bitwidth available.
    Line 20: Signal 'g' may overflow as no enough bitwidth available.
    Line 62: Signal 'c' may overflow as no enough bitwidth available.
    Line 68: Signal 'c' may overflow as no enough bitwidth available.
```



```
Incomplete Sensitivity List:
    Invalid entry in lines: Line 12: Incomplete sensitivity list: Missing signals c in block starting at line 12.
    Invalid entry in lines: Line 18: Incomplete sensitivity list: Missing signals b, g in block starting at line 18.
    Invalid entry in lines: Line 24: Incomplete sensitivity list: Missing signals b in block starting at line 24.
    Invalid entry in lines: Line 29: Incomplete sensitivity list: Missing signals a, c in block starting at line 29.
    Invalid entry in lines: Line 35: Incomplete sensitivity list: Missing signals c in block starting at line 35.
    Invalid entry in lines: Line 44: Incomplete sensitivity list: Missing signals b0, b1, case, endcase, c in block s
    Invalid entry in lines: Line 54: Incomplete sensitivity list: Missing signals d, e in block starting at line 54.
    Invalid entry in lines: Line 60: Incomplete sensitivity list: Missing signals b, c in block starting at line 60.
    Invalid entry in lines: Line 86: Incomplete sensitivity list: Missing signals b0, c, b1, case, endcase, d, b00 in
```



```
Lint Report for indexout.v generated at: 2024-12-24 00:43:36

Arithmetic Overflow:
    Line 23: Signal 'i' may overflow as no enough bitwidth available.
Undefined Register Usage:
    Line 23: Register 'i' is undefined
    Line 23: Register 'i' is undefined
Uninitialized Register Case:
    Line 23: Uninitialized register 'i' used before initialization. Lines: 23.
Array Index Out of Bounds:
    Line 15: Array 'memory' index 32 out of bounds. Valid range: [0:31].
    Line 16: Array 'large_array' index 130 out of bounds. Valid range: [0:127].
    Line 17: Array 'test_wire' index 8 out of bounds. Valid range: [0:7].
    Line 20: Potential out of bounds access for array 'memory' with variable index 'index'.
    Line 24: Potential out of bounds access for array 'memory' with variable index 'i'.
```

Conclusion:

The Verilog Linter is designed to make life easier for hardware designers by catching common mistakes in Verilog code before they become bigger problems. By automating checks for issues like arithmetic overflows, undefined registers, inferred latches, and sensitivity list errors, it helps ensure that designs are both reliable and efficient. The tool not only saves time but also reduces the frustration of manually debugging tricky issues. Its modular design makes it easy to improve and add new features in the future, ensuring it stays relevant as designs become more complex. Overall, this project highlights how tools like the Verilog Linter can simplify the design process, promote better coding practices, and help create higher-quality hardware systems.