EDA Project (Phase 2)

Verilog Lint

# Introduction

This project is aims to create static Verilog Design Checker, the checker will take Verilog (DUT) no Test Bench, and statically points to following List of Violations:

* Arithmetic Overflow
* Unreachable FSM State
* Un-initialized Register
* Multi-Driven Bus
* Non Full/Parallel Case

In this document we will go on each violation and illustrates how it it’s implemented and its limitations.

# Assumptions and Limitations

1. This linter must run after syntax is checked (if the syntax is incorrect it might raise errors)
2. Errors like assigning values to input signal and assigning wires in blocks ( might cause errors in the lint)
3. Assignment only works with = sign , <= sign is not working
4. All case conditions must start with begin and end with end

# Violations

## Arithmetic Overflow

### Illustration

First, we need to illustrate Assignment Overflow. Assignment Overflow mainly arises when you are trying to assign variable of greater size to smaller one in more abstract way it’s when the size of the right hand greater than the size of the left hand of an assign statement .

Ex : a[4:0] b[2:0] so when you say b = a there is an Assignment Overflow as b size is smaller than a

Arithmetic Overflow is subclass of Assignment Overflow but it happens with arithmetic operation.

Ex : a[4:0] b[0:0] so when you say a = a+b an Arithmetic Overflow might arise the output may be of size 6 if a = 1’b1111 and b = 1b‘1

### How it is implemented

1. Parse the Verilog code and get all the variable sizes
2. Parse all assign statements
3. Compute the left hand size and right hand size of each statement using some rules we will define downward
4. If right hand size > left hand size raise an Assignment Overflow Warning on this statement

### Useful Rules

This rules helps you to replace each arithmetic or logic operation with their worst case size scenario.

A+B : result size = max( size(A) , size(B) ) + 1

A\*B : result size = size(A) + size(B)

A-B : result size = size(A)

A/B : result size = Size(A)

All logic operator ( & , | , ^) result size = max( size(A) , size(B) )

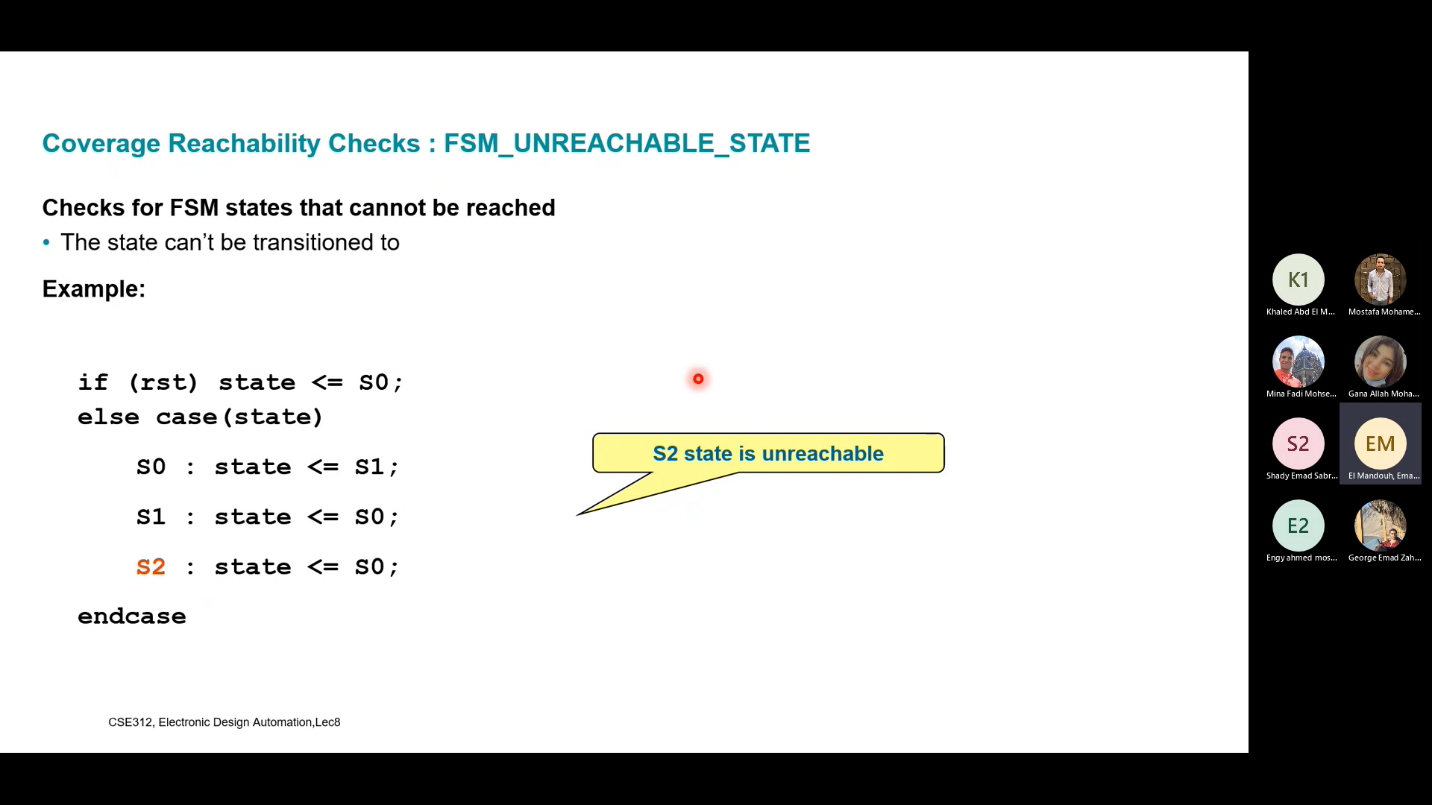
Using this rules you can compute the size of any statement contains arithmetic and logic operators.

### Limitations

1. The lint assumes all operation has same precedence.
2. The lint can’t handle nested statements like statements with brackets inside it (might arise errors if you tried it)

## Unreachable FSM State

### Illustration



Unreachable FSM State happens when a statement will be never reached as the state variable will never be assigned this value. In more abstract way when you have case statement on a variable and this case condition will never be reached.

### How it is implemented

1. Parse Verilog code and get all register that has initial values and save their initial values
2. Parse Verilog code and get all case statements
3. If you find a case it’s condition is register in your initialized registers array then go to next step
4. propagate through the case using the initial value
5. find the statements that change the state variable inside your satisfied condition statements you have four conditions
   1. there is no statement that changes your state variable inside this condition then you are in deadlock state
   2. there is a statement that changes your state variable but it changes it to a state that you already visited so you are looping (you need to have an array that tracks the states that you have already visited)
   3. there is a statement that changes your state variable but it changes it to same variable as the one in the condition so you are in deadlock state which transit only to itself
   4. there is a statement that changes your state variable and changes it to new state then go back to step 4 using the new state instead initial state and loop until one of the above conditions break the loop

### Limitations

1. register must be initialized in the same statement while created not inside an initial block like (reg[3:0] name = 4’b0000)
2. if case condition have some if else statements the lint might not behave correctly

## Un-initialized Register

### Illustration

Any bit that is defined in the code but will never get a value in the code.

### How it is implemented

1. Parse Verilog code and get all variable sizes
2. Parse Verilog code and get all assignment statements
3. Create a key-value Data Structure which contains your variables as keys and an array[size(variable)] filled with zeros correspond to each key ex: reg[3:0] a; {“a”:[0,0,0,0,0]}
4. Then for each assign statement compute change the corresponding bits in your dictionary to one
5. After looping on all assignment statements and filling your dictionary find any bit that has zero and this bit will never be assigned value in the code

## Multi-Driven Bus

### Illustration

Any bit that is assigned values more than one time concurrently.

### How it is implemented

1. Parse Verilog code and get all variable sizes
2. Parse Verilog code and get all concurrent assignment statements.
3. Create a key-value Data Structure which contains your variables as keys and an array[size(variable)] filled with zeros correspond to each key ex: reg[3:0] a; {“a”:[0,0,0,0,0]}
4. Then for each assign statement compute change the corresponding bits in your dictionary to their previous value + 1
5. After looping on all assignment statements and filling your dictionary find any bit that has value greater than one so this bit is assigned multiple concurrent statements.

## Non Full/Parallel case

### Illustration

A case statement in Verilog is said to be a full case when it specifies the output for every value of the input. In the context of synthesizable code, this means specifying the output for all combinations of zeros and ones in the input. Case items involving X’s and Z’s are ignored by synthesis tools.

The significance of full case statements is that case statements that are not full infer latches unless a synthesis pragma – *full case* is used. A latch is inferred because it matches the behavior in simulation which is to remember the previous output when input changes to an unspecified value. In most cases, such latches are unintended and undesirable.

A case statement in Verilog is said to be a parallel case when it isn't possible for multiple case items to be equal in value to the case-expression.

The significance of parallel case statements is that case statements that are not parallel infer priority logic during synthesis. This is done to match the behavior in simulation which is to assign the output according to the first matching case-item only (even when there are multiple case-item matches).

### How it is implemented

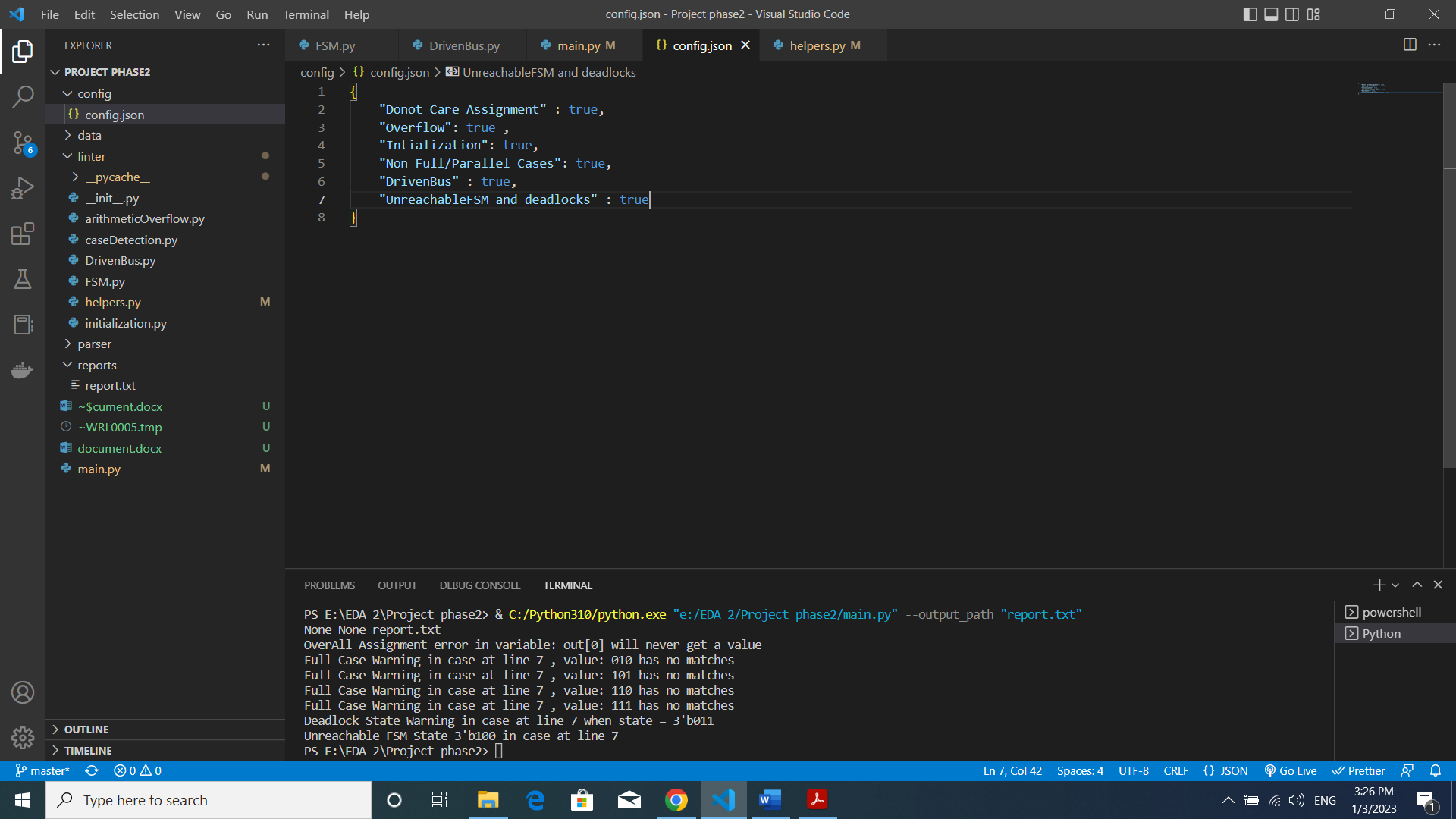
1. Parse Verilog code and get all variable sizes
2. Parse Verilog code and get all case
3. For each case get all combinations from its variable size like if we have case(a) and a [1:0] then all our combinations are (00,01,10,11)
4. Create a dictionary that contains all combinations as keys ex: {“00”: 0, “01”:0,”10”:0,”11”:0}
5. Loop over each condition and go to its corresponding key then add it by 1 you have two situations
   1. The condition contains don’t care like 0x then you need to convert it to its corresponding (01,00) then search the dictionary and add their values by 1
   2. The condition doesn’t contain don’t care like 00 then search the dictionary and add its values by 1
6. Loop over each key and if the key has value = 0 then the case isn’t full because that key and if the key has value greater than 1 then the case isn’t parallel because that key.

### Limitations

1. Only used in cases with constant conditions like case(state[2:0]) begin 3’b000: , 3’b001: , 3’b010:.
2. You can’t nest multiple condition in the same statement like case(state[2:0]) begin 3’b000, 3’b001, 3’b010: . this will cause error.
3. Each condition must start with begin and end with end

# How to use the program

## Config File



You can config what you want to be analyzed and what not by setting it to true and false in the config

## Arguments

python main.py --output\_path "report2.txt" --file\_path "data/testfile6.v" --config\_path "config/config.json"

output\_path is the generated report path default is reports/report.txt

file\_path is the path of Verilog file you want to run analysis on

config\_path is the path of your configuration file

Also , you can run executable file if you don’t have python installed

main.exe --output\_path "reportexe.txt" --file\_path "data/testfile3.v" --config\_path "config/config.json"

# Test cases

## Test case 1:

Verilog code with some initialization errors.

File can be found in data/testfile1.v

Report can be found in reports/report1.txt

## Test case 2:

Verilog code with some initialization errors and two cases one if full and parallel and the other isn’t full and parallel.

File can be found in data/testfile2.v

Report can be found in reports/report2.txt

## Test case 3:

Verilog code with some initialization errors and don’t care assignment error also has some assignment overflow errors.

File can be found in data/testfile3.v

Report can be found in reports/report3.txt

## Test case 4:

Verilog code with bus conflict and some initialization errors and don’t care assignment error also has some assignment overflow errors.

File can be found in data/testfile4.v

Report can be found in reports/report4.txt

## Test case 5:

Verilog code with unreachable FSM state , deadlock state also Non-full case and overall assignment error.

File can be found in data/testfile5.v

Report can be found in reports/report5.txt