TESTBENCH GENERATOR with Python

Lenin Silva Gutiérrez
Jesús Enriquez Gaytán
Eduardo Hernández Polo

Purpose

Perform a python script to auto-generate a Testbench for a Verilog design. Said testbench should be ready to test the given module in EDA Playground without modifying anything.

Project brief

The project consists of a python script to generate a Testbench from a .sv file. Additionally, it can be simulated in the EDAPlayground web for mere verification.

The script begins by opening and acquiring the top module name from the .sv file. It can detect both defined parameters and input, output, and in-out signals with their own name and vector size.

In the terminal, you can interact with the script by introducing the iterator to generate as much stimulus as you want. Also, you can choose up to three different stimulus options for each input to test, listed down below:

1. Random number.
2. Ascending counter.
3. Descending counter.

Once the process finishes, in the folder where the .sv file resides, a new file will be created with the Testbench template.

Issue list

The classification of the different structures in .sv file, for example:

a. Parameter.
b. Input, output, inout.

c. Clk and rst to discriminate if the design is sequential or combinatory.
d. The generator of the batch test.

Design Documentation

The proposed design for the testbench generator script is shown in Fig 1. This script opens a .sv file and extracts all the information needed using regular expressions, e.g., Module name, parameters, inputs, outputs, in-outs with both name and vector size. Also initialize and generate 'clk' and 'rst' signals if detected on the design file.

Once the extraction is done and all the information is stored for further processing, the script asks the user to type in an 'Iterator' value to generate a stimulus for input simulation. Furthermore, let the user choose among three stimulus options for each input, i.e, random values, up-counter and down-counter.

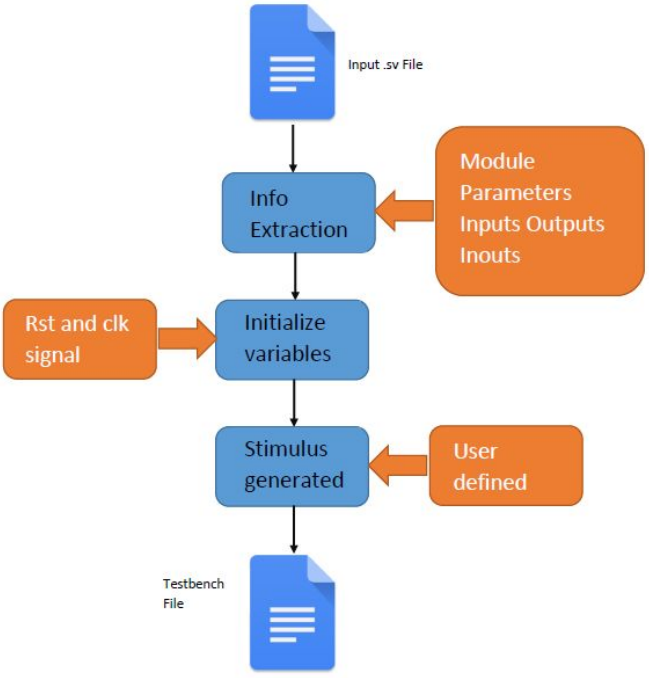


Fig 1. Approach to Testbench Generator.

Right after that, the script gives Testbench format to all inputs/outputs/in-outs, reg or wire, depending the case. Then, initializes all variables used in the design file in zero ('clk' in zero and 'rst' in one, if detected) and generates the 'clk' signal. In addition, generates all stimulus needed to simulate inputs based on previous user-defined values.

Finally, the script creates and opens a new .sv file named with the design file name plus "_tb" and writes down a general Testbench template with all the variables and stimulus included within.

Regex usage

The information extraction was done using regular expressions and groups. Three regex were defined to identify the module name, the parameters and the signals of the module provided. Also, another regex was added to detect comments in the Verilog file and ignore them.

The following regex was used to detect comments:

```
r'\/\/[^\n]*|\/\*((?!\*\/).)*\*\/'
```

Fig 2 shows the working example of the regex detecting multiple line and single line comments. It is important to note that for this regex to work, the re.DOTALL flag must be included in Python so that the dot matches every character, including newlines.

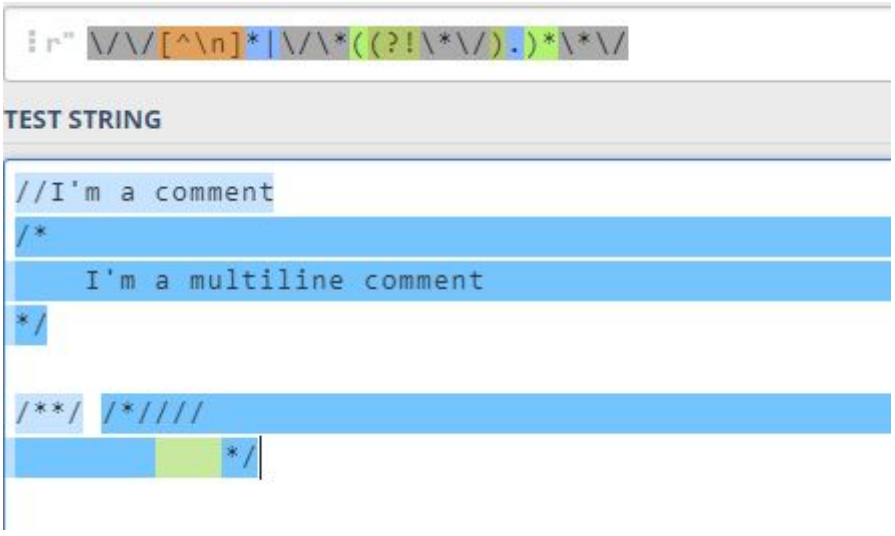


Fig 2. Comments regex detecting single and multiple line comments

To detect the module's name, the following regex was used:

```
module\s+([_a-zA-Z]\w*)
```

Fig 3 shows how the regex matches the module name and assigns it as group 1 in the regex.

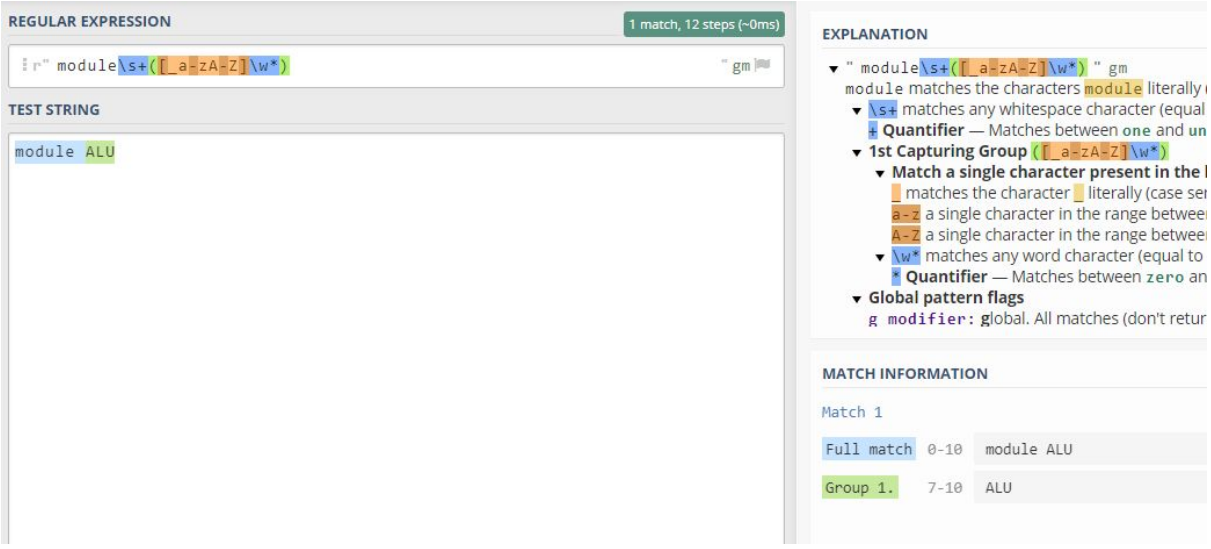


Fig 3. Regex for the module name demonstration.

The following regex was used to detect the parameters defined in the module

```
((parameter)\s+(\w*)\s*\=\s*((\d+\'(b|h|d))?\w+))
```

Fig 4 shows the regex working. From that image, it is seen that each parameter becomes a full match, which has six groups inside, which are defined as follows:

- 1st group → Full match
- 2nd group → word *parameter*
- 3rd group → name of the parameter
- 4th group → bit size of the parameter

The other groups are not relevant for identification purposes.

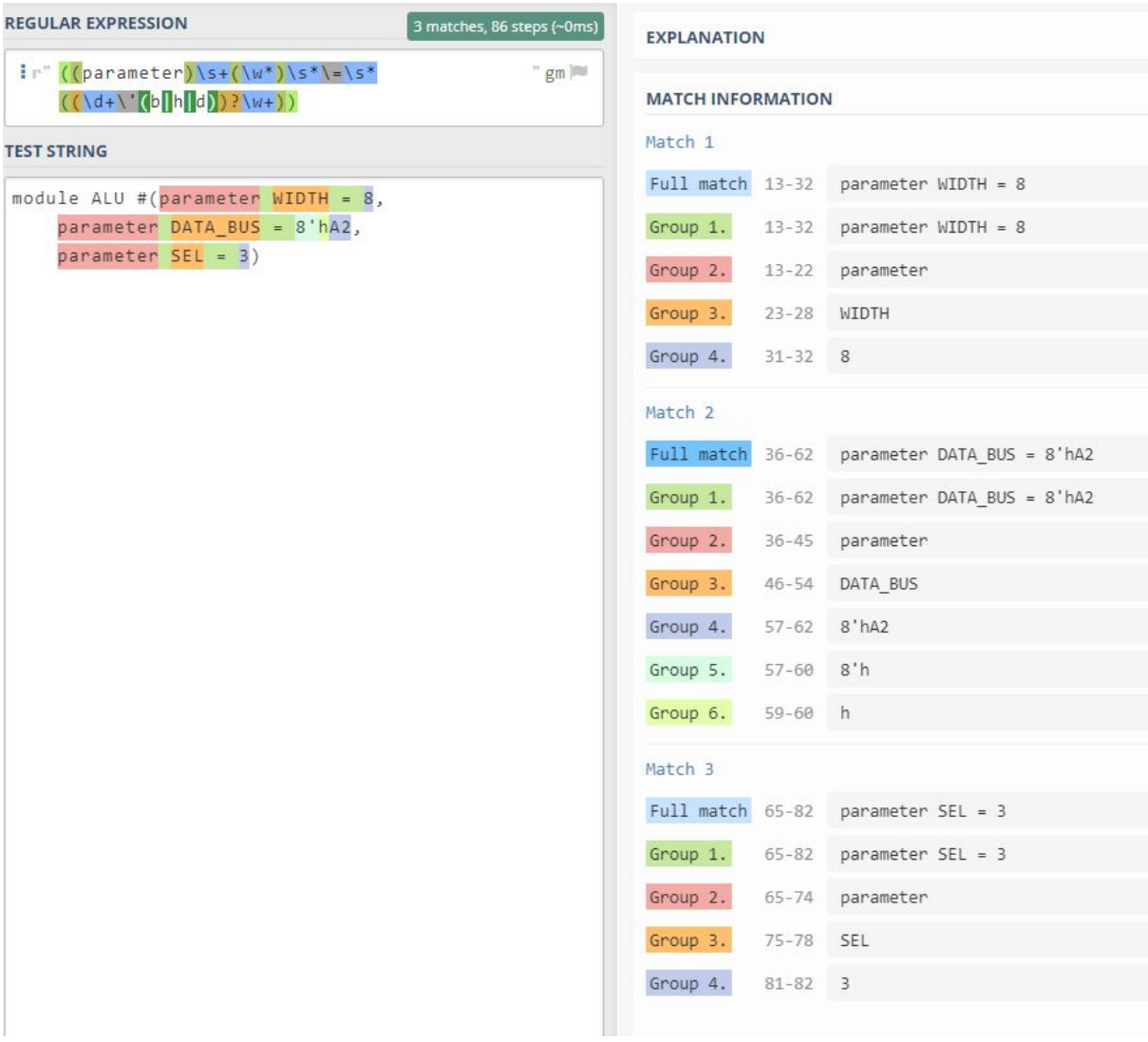


Fig 4. Demonstration of parameters' regex.

Finally, the last regex was to detect inputs, outputs and inouts in the module, whether they are defined in the module declaration or between the code.

```
(input|output|inout)(\s+(reg|logic))?(\s*\[[^\]]+\]\s*|\s+)((?!input|output|inout|reg|logic)[_a-zA-Z]\w*(,\s*(?!input|output|inout|reg|logic)[_a-zA-Z]\w*)*)
```

Since this regex is the most complex and largest, it will be broken down into its different components. First, the working example is shown in Fig. 5

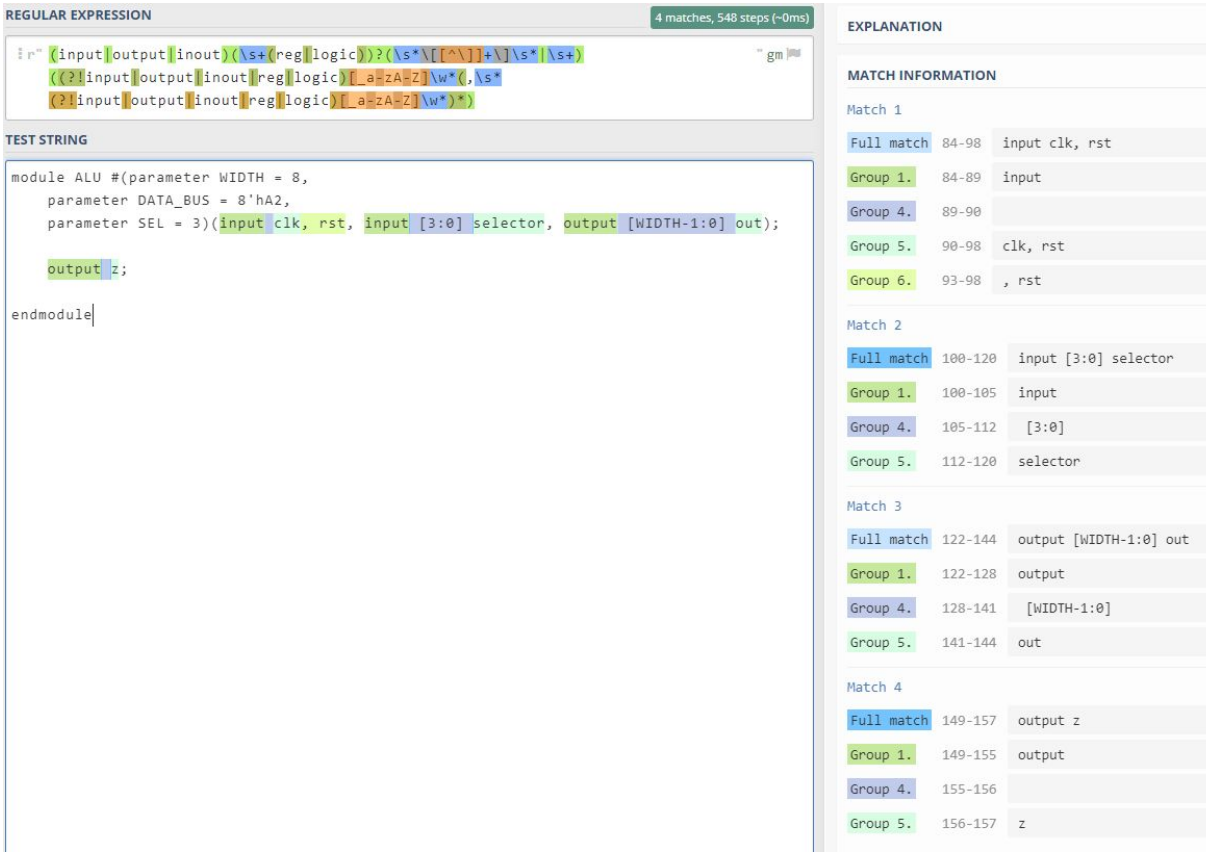


Fig. 5 Input, output and in-out regex working example.

From this image, it is seen that there are around six groups detected with the regex. Each group will be broken down.

1st group

The first capturing group is in charge of detecting the words *input, output* or *inout*, with the regex **(input|output|inout)**

2nd and 3rd groups

The second group finds the words reg or logic with the regex **(\s+(reg|logic))?**. It includes the **?** modifier to specify that this group can exist zero or one time. It also includes at least one white space before the words. The third capturing group is included inside the second, so it identifies only the words, without whitespaces.

4th group

The fourth group finds the bus size of the variables with the regex **(\s*\[[^\]]+\]\s*|\s+)**. It detects either anything enclosed in square brackets or at least one whitespace, meaning that this group has the bus width or a whitespace if the variable is just one bit long.

5th group

This group is the one in charge of identifying the variables names. The regex is the following:

((?!input|output|inout|reg|logic)[_a-zA-Z]\w*(,\s*(?!input|output|inout|reg|logic)[_a-zA-Z]\w*)*)

This regex identifies variable names with the regex **[_a-zA-Z]\w*,** so that every variable starting with a letter or underscore, followed by any alphanumeric character is accepted. However, it needs to detect whether the word is reserved (like input or output), so that it does not classify it as another variable. That is why it includes a negative lookahead regex **(?!input|output|inout|reg|logic)** in charge of checking that the variable found is not an input, output, inout, reg or logic word. Finally, to detect multiple variables separated by comma, a sixth group inside this group is added, with a comma followed by whitespaces and the same regex to detect variable names, with negative lookahead included.

So, with all these groups, it was possible to detect all the inputs, outputs and inouts with the function **re.findall()**, which returned a list of tuples, where each tuple contained every capturing group with its corresponding match. With that list of tuples, it was possible to iterate over it, check each group, separate all the variables in that tuple with a **split(“,”)**, and add them to the corresponding input, output or inout dictionary.

Demonstration

In order to demonstrate how the Testbench generator works, we show the complete Testbench file generation using a specific Verilog file (.sv).

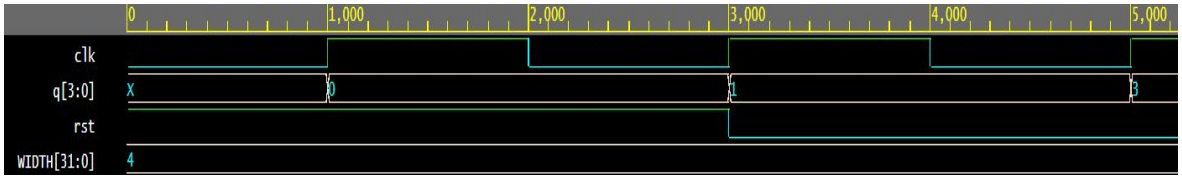
--Code version 1--

a)

```
// Code your design here
module gray_ctr
  #(parameter WIDTH=4)
  (input  clk,
    input  rst,
    output  [WIDTH-1:0] q);

  reg [WIDTH-1:0] counter;

  always @ (posedge clk) begin
    if (rst) begin
      counter <= 0;
    end
    else begin
      counter <= counter + 1;
    end
  end

  always begin
  for (int i=1; i < WIDTH; i = i + 1) begin
    q[i-1] = counter[i]^counter[i-1];
  end
  q[WIDTH-1] = counter[WIDTH-1];
  end


endmodule
```

b)

```
`timescale 1ns/1ps

module gray_ctr_tb;

    parameter WIDTH=4;

    //Creación de regs y wires
    reg  clk;
    reg  rst;

    wire [WIDTH-1:0] q;

    //Instanciar el top
    gray_ctr UUT(.*);

initial
  begin
    $dumpfile("gray_ctr_tb.vcd");
    $dumpvars (1, gray_ctr_tb);

    clk = 0;
    rst = 1;

    # 3
    rst = 0;


    #4
    $finish;

   end

 always forever #1 clk = ~clk;
```



c)

Table 1. a) Verilog main module. b) Testbench generated. c) EP waveform on EDA Playground.

--Code version 2--

This time around, we demonstrate the new version with another example, adding some of the changes we made.

a)

```
// Code your design here
module register_file (
        input logic         rst,
        input logic         clk,
        input logic   [4:0] rs_addr,
        input logic   [4:0] rt_addr,
        input logic   [4:0] rd_addr,
        input logic  [31:0] rd_w_data,
        input logic         reg_write,
        output logic [31:0] rs_data,
        output logic [31:0] rt_data
    );

  logic [31:0] reg_mem [0:31];

    integer i;

    always_ff @(posedge clk or posedge rst)
        begin
            if(rst)
                for(i = 0; i <= 31; i = i+1)
                    reg_mem[i] <= 0;
            else if(reg_write & rd_addr != 0)
                reg_mem[rd_addr] <= rd_w_data;

        end

  /*
    always_ff @(posedge clk)
      begin
        rs_data <= reg_mem[rs_addr];
        rt_data <= reg_mem[rt_addr];
      end
  */
  assign rs_data = reg_mem[rs_addr];
  assign rt_data = reg_mem[rt_addr];

endmodule
```

b)

```
// Create Date:  11/11/2020, 10:57:57
// Project Name: register_file

`timescale 1ns/1ps

module register_file_tb;

    //Creación de regs y wires
    reg  rst;
    reg  clk;
    reg [4:0] rs_addr;
    reg [4:0] rt_addr;
    reg [4:0] rd_addr;
    reg [31:0] rd_w_data;

    wire [31:0] rs_data;
    wire [31:0] rt_data;

    //Instanciar el top
    register_file UUT(.*);

initial
  begin
    $dumpfile("register_file_tb.vcd");
    $dumpvars (1, register_file_tb);

    rst = 1;
    clk = 0;
    rs_addr = 0;
    rt_addr = 0;
    rd_addr = 0;
    rd_w_data = 0;
    #3
    rst = 0;
    for(integer i = 0; i < 10; i++) begin
        #2
        rs_addr = $urandom(90732);
        rt_addr = i;
        rd_addr = 9-i;
        rd_w_data = $urandom(451);
    end
    #4
    $finish;
   end
 always forever #0.5 clk = ~clk;
endmodule
```

```
PS C:\Users\vsiwr\Documents\Proyectos\Pre-Silicio\top-tb-generator> python .\Python\main.py .\Tests\rf.sv
Welcome to the testbench generator!

For input [4:0] rs_addr, what do you want to do?
1 -> Random signal generation
2 -> Ascending counter
3 -> Descending counter
Option: 1

For input [4:0] rt_addr, what do you want to do?
1 -> Random signal generation
2 -> Ascending counter
3 -> Descending counter
Option: 2

For input [4:0] rd_addr, what do you want to do?
1 -> Random signal generation
2 -> Ascending counter
3 -> Descending counter
Option: 3

For input [31:0] rd_w_data, what do you want to do?
Option: 3

For input [31:0] rd_w_data, what do you want to do?
1 -> Random signal generation
2 -> Ascending counter
3 -> Descending counter
Option: 1
Done
```

c)

Table2. a) Verilog main module. b) Testbench generated. c) Terminal prompt generated without flags.

a)

```verilog
// Code your design here
module register_file (
        input logic         rst,
        input logic         clk,
        input logic   [4:0] rs_addr,
        input logic   [4:0] rt_addr,
        input logic   [4:0] rd_addr,
        input logic  [31:0] rd_w_data,
        input logic         reg_write,
        output logic [31:0] rs_data,
        output logic [31:0] rt_data
    );

  logic [31:0] reg_mem [0:31];

    integer i;

    always_ff @(posedge clk or posedge rst)
        begin
            if(rst)
                for(i = 0; i <= 31; i = i+1)
                    reg_mem[i] <= 0;
            else if(reg_write & rd_addr != 0)
                reg_mem[rd_addr] <= rd_w_data;

        end

  /*
    always_ff @(posedge clk)
      begin
        rs_data <= reg_mem[rs_addr];
        rt_data <= reg_mem[rt_addr];
      end
  */
  assign rs_data = reg_mem[rs_addr];
  assign rt_data = reg_mem[rt_addr];

endmodule
```

b)

```verilog
// Create Date:  11/11/2020, 11:01:33
// Project Name: register_file

`timescale 1ns/1ps

module register_file_tb;

    //Creación de regs y wires
    reg  rst;
    reg  clk;
    reg [4:0] rs_addr;
    reg [4:0] rt_addr;
    reg [4:0] rd_addr;
    reg [31:0] rd_w_data;

    wire [31:0] rs_data;
    wire [31:0] rt_data;

    //Instanciar el top
    register_file UUT(.*);

initial
  begin
    $dumpfile("register_file_tb.vcd");
    $dumpvars (1, register_file_tb);

    rst = 1;
    clk = 0;
    rs_addr = 0;
    rt_addr = 0;
    rd_addr = 0;
    rd_w_data = 0;
    #3
    rst = 0;
    for(integer i = 0; i < 32; i++) begin
        #2
        rs_addr = $urandom(16594);
        rt_addr = $urandom(70236);
        rd_addr = $urandom(70129);
        rd_w_data = $urandom(45014);
    end
    #4
    $finish;
   end
 always forever #0.5 clk = ~clk;
endmodule
```

```
PS C:\Users\vsiwr\Documents\Proyectos\Pre-Silicio\top-tb-generator> python .\Python\main.py .\Tests\rf.sv -rf
Welcome to the testbench generator!

Input loop for iterations (default 10): 32
Done
```

c)

Table3. a) Verilog main module. b) Testbench generated. c) Terminal prompt generated with "-rf" flags, all stimuli are random, and the number of iterations is 32..

a)

```
module register_file #(
  parameter ADDR = 5,
  parameter BUS_W = 32
)(
        input         reset,
        input         reloj,
        input   [ADDR-1:0] rs_addr,
        input   [ADDR-1:0] rt_addr,
        input   [ADDR-1:0] rd_addr,
        input  [BUS_W - 1:0] rd_w_data,
        input         reg_write,
        output [BUS_W - 1:0] rs_data,
        output [BUS_W - 1:0] rt_data
    );

  logic [31:0] reg_mem [0:31];

    integer i;

    always_ff @(posedge reloj or posedge reset)
        begin
            if(rst)
                for(i = 0; i <= 31; i = i+1)
                    reg_mem[i] <= 0;
            else if(reg_write & rd_addr != 0)
                reg_mem[rd_addr] <= rd_w_data;

        end

  /*
    always_ff @(posedge clk)
      begin
        rs_data <= reg_mem[rs_addr];
        rt_data <= reg_mem[rt_addr];
      end
  */
  assign rs_data = reg_mem[rs_addr];
  assign rt_data = reg_mem[rt_addr];

endmodule
```

b)

```
// Create Date:  11/11/2020, 11:10:26
// Project Name: register_file

`timescale 100ns/1ps

module register_file_tb;

    parameter ADDR = 5;
    parameter BUS_W = 32;
    //Creación de regs y wires
    reg  reset;
    reg  reloj;
    reg [ADDR-1:0] rs_addr;
    reg [ADDR-1:0] rt_addr;
    reg [ADDR-1:0] rd_addr;
    reg [BUS_W - 1:0] rd_w_data;

    wire [BUS_W - 1:0] rs_data;
    wire [BUS_W - 1:0] rt_data;

    //Instanciar el top
    register_file UUT(.*);

initial
  begin
    $dumpfile("register_file_tb.vcd");
    $dumpvars (1, register_file_tb);

    reset = 0;
    reloj = 0;
    rs_addr = 0;
    rt_addr = 0;
    rd_addr = 0;
    rd_w_data = 0;
    #3
    reset = 1;
    for(integer i = 0; i < 32; i++) begin
        #2
        rs_addr = $urandom(14911);
        rt_addr = $urandom(30397);
        rd_addr = $urandom(81269);
        rd_w_data = 31-i;
    end
    #4
    $finish;
   end
 always forever #0.5 reloj = ~reloj;
endmodule
```

```
PS C:\Users\vsiwr\Documents\Proyectos\Pre-Silicio\top-tb-generator> python .\Python\main.py .\Tests\rf.sv -fcst
Welcome to the testbench generator!

Define new name of your clock: reloj

Define the new name of your reset: reset

Is it active high? [y/n]: n

For input [ADDR-1:0] rs_addr, what do you want to do?
1 -> Random signal generation
2 -> Ascending counter
3 -> Descending counter
Option: 1

For input [ADDR-1:0] rt_addr, what do you want to do?
1 -> Random signal generation
2 -> Ascending counter
3 -> Descending counter
Option: 1

For input [ADDR-1:0] rd_addr, what do you want to do?
1 -> Random signal generation
2 -> Ascending counter
3 -> Descending counter
Option: 1

For input [BUS_W - 1:0] rd_w_data, what do you want to do?
1 -> Random signal generation
2 -> Ascending counter
3 -> Descending counter
Option: 3

Input loop for iterations (default 10): 32

Set time unit and time precision for timescale: 100ns/1ps
Done
```

c)

Table4. a) Verilog main module with parameters. b) Testbench generated include parameters. c) Terminal prompt generated with "-fcst" flags. The user defines the name of both clock and reset (active low) signals, the number of iterations is 32, and sets a user-defined timescale.

In the next picture, the help menu is shown. The user can access this menu by typing the argument --help.

```
PS C:\Users\vsiwr\Documents\Proyectos\Pre-Silicio\top-tb-generator> python .\Python\main.py --help

    python3 main.py [OPTIONS] [FILENAME]

    Options:

    -r --> All variables are assigned a random number $urandom() in every iteration
    -a --> Variables are assigned a number that increases by 1 with each iteration
    -d --> Variables are assigned a number that starts at the for loop limit and decreases by 1 every iteration
    -t --> Override default timescale of 1ns/1ps
    -s --> Override default reset name (rst) and active high
    -c --> Override default clock name (clk)
    -f --> Override default number of iterations (10)

    With no option, the user will be prompted to select the value to assign for each variable, the loop iterations
    will be set to 10, the clock signal is expected to be named "clk" and the reset signal is expected to be "rst"
    and active high, and the timescale is set to 1ns/1ps.
```

Fig. 6 Help menu in the terminal.

Code

--Code Version 1--

The first approach to create the Testbench template.

<u>displayMenu.py</u>

```
# Function to generate stimulus based on User-defined iterator, by default generates up to 10 stimulus
def selectForIterations():
    forIt = input("\nInput loop for iterations (default 10): ")
    return int(forIt) if forIt.isnumeric() else 10

# Function to display a menu to the user and select what kind of stimulus wants to generate
def displayMenu(varTuple):

    print("\nFor input %s %s, what do you want to do?" %
          (varTuple[1], varTuple[0]))
    print("1 -> Random signal generation\n2 -> Up-counting\n3 -> Descending counter")
    opt = input("Option: ")
    while opt != "1" and opt != "2" and opt != "3":
        print("Invalid option!\n")
        print("1 -> Random signal generation\n2 -> Up-counting\n3 -> Descending counter")
        opt = input("Option: ")

    if opt == "1":
        return "random"
```

```
    elif opt == "2":
        return "up"
    else:
        return "down"
```

strFuncs.py

```
# var_struct --> (name, size, type, funcType)

# Function to generate all (regs) inputs and/or inouts within the testbench
def generateInputTb(input_dicc, inout_dicc):
    s = ""
    for i in input_dicc.values():
        s += "\treg %s %s;\n" % (i[1], i[0])
    for i in inout_dicc.values():
        s += "\treg %s %s;\n" % (i[1], i[0])
    return s

# Function to generate all (wires) outputs within the testbench
def generateOutputTb(output_dicc):
    s = ""
    for o in output_dicc.values():
        s += "\twire %s %s;\n" % (o[1], o[0])
    return s

# Generates stimulus values for inputs to simulate
def generateMainSequence(input_dicc, forIt):
    exists = False
    s = "\t\tfor(integer i = 0; i < %d; i++) begin\n\t\t\t#2" % forIt
    for varTuple in input_dicc.values():
        if varTuple[0] != 'clk' and varTuple[0] != 'rst':
            exists = True
            # busSize = int(varTuple[1].split(":")[0][1:]) + \
            #     1 if varTuple[1].strip() != "" else 1
            if varTuple[3] == 'random':
                s += f"\n\t\t\t{varTuple[0]} = $urandom();"
            elif varTuple[3] == 'up':
                s += f"\n\t\t\t{varTuple[0]} = i;"
            elif varTuple[3] == "down":
                s += f"\n\t\t\t{varTuple[0]} = {forIt-1}-i;"
    s += "\n\t\tend"
    return s if exists else ""
```

```
# Function to initialize input variables within the testbench
def variableInit(input_dicc):
    s = ""
    for varTuple in input_dicc.values():
        if varTuple[0] != 'clk' and varTuple[0] != 'rst':
            s += f"\n\t\t{varTuple[0]} = 0;"
        # if varTuple[0] != 'clk' and varTuple[0] != 'rst':
        #     busSize = int(varTuple[1].split(":")[0][1:]) + \
        #         1 if varTuple[1].strip() != "" else 1
        #     if varTuple[3] == 'random':
        #         s += generateRandom(busSize, varTuple[0])
            # elif varTuple[3] == 'up':
            #     s+=generateAscending()
            # elif varTuple[3] == 'down':
            #     s+=generateDescendign()
    return s

# Function to create the Verilog testbench template with module, parameters (if so), inputs/outputs
variables, initialization and stimulus
def getTBString(moduleName, paramsStr, regStr, wireStr, hasClk, hasRst, varInit, mainSequence):

    rstInit = """rst = 1;

    # 3
    rst = 0;"""

    return f"""
`timescale 1ns/1ps

module {moduleName}_tb;
{paramsStr}

    //Creación de regs y wires
{regStr}
{wireStr}
    //Instanciar el top
    {moduleName} UUT(.*);

initial
  begin
    $dumpfile("{moduleName}_tb.vcd");
```

```
   $dumpvars (1, {moduleName}_tb);
{varInit}
   {"clk = 0;" if hasClk else ""}
   {rstInit if hasRst else ""}

{mainSequence}
   #4
   $finish;

  end

 {"always forever #1 clk = ~clk;" if hasClk else ""}


endmodule
   """
```

main.py

```
import sys
import re
from strFuncs import *
from displayMenu import *

# Global regex and variables
re_module_name = r'module\s+([_a-zA-Z]\w*)'


1st group --> input|ouput|inout
3rd group --> logic | reg
4th group --> bus size
5th group --> variables separated by coma


re_inout =
r'(input|output|inout)(\s+(reg|logic))?(\s*\[[^\]]+\]\s*|\s+)((?!input|output|inout|reg|logic)[_a-zA-Z]\w*(,\s*(
?!input|output|inout|reg|logic)[_a-zA-Z]\w*)*)'

'''
1st group --> param_name
2nd group --> param_size
'''
```

```
re_parameters = r'((parameter)\s+(\w*)\s*\=\s*((\d+\'(b|h|d))?\w+))'  #

'''
key --> variable name
var_struct --> (name, size, type, funcType)
{
    key: var_struct
}

'''
# Global dictionaries
input_dicc = {}
output_dicc = {}
inout_dicc = {}

if __name__ == "__main__":

    if len(sys.argv) < 2:
        print("Missing input arguments!")
        sys.exit(0)
    # Argument provided by the console with the file to open
    filename = sys.argv[1]

    # Open the design file
    f = open(filename, 'r')
    text = f.read()  # Read it as a string
    f.close()  # Close the file

    print("Welcome to the testbench generator!")

    # Get the module name
    moduleName = re.findall(re_module_name, text)[0]
    # Get all parameters
    params_list = re.findall(re_parameters, text)
    # Get all the inputs and outputs in the text
    inout_list = re.findall(re_inout, text)

    # print(params_list)
```

```
# Give proper format to parameters within the testbench
    # Iterate over parameters
    paramsStr = ""
    for par in params_list:
        paramsStr += "\n\t" + par[0] + ";"

    # Flags for clk and rst (verify if sequential or combinational verilog design)
    hasClk = False
    hasRst = False

    # Iterate over the inputs and outputs
    for m in inout_list:
        '''
        Groups in the regex match
        1st group --> input|ouput|inout
        3rd group --> logic | reg
        4th group --> bus size
        5th group --> variables separated by coma
        var_struct --> (name, size, type, funcType)
        '''
        varList = m[4].split(",")  # Get variables list
        for var in varList:  # Iterate over the variables list

            # Create the tuple to save the variable info
            varTuple = [var.strip(), m[3].strip(), m[2], '']
            if not hasClk:
                hasClk = varTuple[0] == 'clk'
            if not hasRst:
                hasRst = varTuple[0] == 'rst'

            # Save the tuple in the appropriate dictionary
            if(m[0] == "input"):
                if varTuple[0] != 'clk' and varTuple[0] != 'rst':
                    varTuple[3] = displayMenu(varTuple)
                input_dicc[varTuple[0]] = varTuple
            elif(m[0] == "output"):
                output_dicc[varTuple[0]] = varTuple
            else:  # inout
                inout_dicc[varTuple[0]] = varTuple

    # User type in iterator for simulation values
    forIt = selectForIterations()
```

```
# Generate variable delcarations in SystemVerilog format
regStr = generateInputTb(input_dicc, inout_dicc)
wireStr = generateOutputTb(output_dicc)
varInit = variableInit(input_dicc)
mainSequence = generateMainSequence(input_dicc, forIt)

# Open the test bench file in utf8 encoding
tbName = filename[0:len(filename)-3]+"_tb.sv"
tb = open(tbName, 'w', encoding='utf8')
# Write the file with the string formatted appropriately
tb.write(getTBString(moduleName, paramsStr, regStr,
            wireStr, hasClk, hasRst, varInit, mainSequence))
# Close the file
tb.close()
print("Done")
```

--Code Version 2--

Addition of new modules to improve user interaction with the script. Some changes were made to the code for layout improvements and codeline reduction.

- displayMenu.py
  - timescale() → allows the user to define the timescale and override the default (1ns/1ps).
  - getClk() → lets the user select the clock signal name and override the default (clk).
  - getRst() → prompts the user to select the reset signal name and if it's active high or active low (default is “rst” and active high).
  - printHelp() → display a menu for fast testbench set up.
- strFuncs.py
  - generateMainSequence() → random seed generation for $urandom() stimulus.
- main.py
  - Remove comments from the file with a new regex parameter.
  - Get the current time to display it on the testbench’s information header as a comment.
  - Read the command line arguments proposed for the user to simplify the generator of the testbench. It allows the following flags:

- -r → sets all signals stimulus as random
- -a → sets all signals stimulus with an ascending counter inside the for loop
- -d → sets all signals stimulus with a descending counter inside the for loop
- -t → indicates that the user wants to override the default timescale
- -c → indicates that the user wants to override the default clock name (clk)
- -s → indicates that the user wants to override the default reset name (rst) and active high
- -f → indicates that the user wants to override the default number of iterations (10)
- --help → prints a simple manual with the options and their descriptions and stops execution without building the testbench

displayMenu.py

```
# Function to set a timescale
def timescale():
    ts = input("\nSet time unit and time precision for timescale: ")
    return ts

# Function to ask user for a desirable clk signal
def getClk():
    clk = input("\nDefine new name of your clock: ")
    return clk

# Function to ask user for a desirable rst signal
def getRst():
    rstName = input("\nDefine the new name of your reset: ")
    active = input("\nIs it active high? [y/n]: ").lower()
    while active != 'y' and active != 'n':
        print("Invalid option!\n")
        active = input("\nIs it active high? [y/n]: ").lower()
    return (rstName, active == "y")
```

```
# Function to generate stimulus based on User-defined iterator, by
default generates up to 10 stimulus
def selectForIterations():
    forIt = input("\nInput loop for iterations (default 10): ")

    return int(forIt) if forIt.isnumeric() else 10

# Function to display a menu to the user and select what kind of
stimulus wants to generate
def displayMenu(varTuple):

    print("\nFor input %s %s, what do you want to do?" %
          (varTuple[1], varTuple[0]))
    print("1 -> Random signal generation\n2 -> Ascending counter\n3 ->
Descending counter")
    opt = input("Option: ")
    while opt != "1" and opt != "2" and opt != "3":
        print("Invalid option!\n")


print("1 -> Random signal generation\n2 -> Ascending counter\n3 ->
Descending counter")
        opt = input("Option: ")

    if opt == "1":
        return "random"
    elif opt == "2":
        return "up"
    else:
        return "down"

# Function to display a help menu for a fast set up
def printHelp():
    print("""
    python3 main.py [OPTIONS] [FILENAME]

    Options:

    -r --> All variables are assigned a random number $urandom() in
every iteration
    -a --> Variables are assigned a number that increases by 1 with
each iteration
```

```
    -d --> Variables are assigned a number that starts at the for loop
limit and decreases by 1 every iteration
    -t --> Override default timescale of 1ns/1ps
    -s --> Override default reset name (rst) and active high
    -c --> Override default clock name (clk)
    -f --> Override default number of iterations (10)

    With no option, the user will be prompted to select the value to
assign for each variable, the loop iterations
    will be set to 10, the clock signal is expected to be named "clk"
and the reset signal is expected to be "rst"
    and active high, and the timescale is set to 1ns/1ps.

    """)
    return
```

strFuncs.py

```
from random import randint
# var_struct --> (name, size, type, funcType)
# Function to generate all (regs) inputs and/or inouts within the
testbench
def generateInputTb(input_dicc, inout_dicc):
    s = ""
    for i in input_dicc.values():
        s += "\treg %s %s;\n" % (i[1], i[0])
    for i in inout_dicc.values():
        s += "\treg %s %s;\n" % (i[1], i[0])
    return s
# Function to generate all (wires) outputs within the testbench
def generateOutputTb(output_dicc):
    s = ""
    for o in output_dicc.values():
        s += "\twire %s %s;\n" % (o[1], o[0])
    return s
```

```
# Generates stimulus values for inputs to simulate
def generateMainSequence(input_dicc, forIt, clk, rst):
    exists = False
    s = "\tfor(integer i = 0; i < %d; i++) begin\n\t\t#2" % forIt
    for varTuple in input_dicc.values():
        if varTuple[0] != clk and varTuple[0] != rst[0]:
            exists = True
            if varTuple[3] == 'random':
                 s+=f"\n\t\t{varTuple[0]}= $urandom({randint(1,100000)});"
            elif varTuple[3] == 'up':
                s += f"\n\t\t{varTuple[0]} = i;"
            elif varTuple[3] == "down":
                s += f"\n\t\t{varTuple[0]} = {forIt-1}-i;"
    s += "\n\tend"
    return s if exists else ""
# Function to initialize input variables within the testbench
def variableInit(input_dicc, clk, rst):
    s = ""
    for varTuple in input_dicc.values():
        s += f"\n\t{varTuple[0]}"
        if(varTuple[0] == rst[0]):
            s += " = 1;" if rst[1] else " = 0;"
        else:
            s += " = 0;"
    return s
# Function to create the Verilog testbench template with module,
parameters (if so), inputs/outputs variables, initialization and
stimulus
def getTBString(date_time, moduleName, paramsStr, regStr, wireStr,
hasClk, hasRst, varInit, mainSequence, scale, clk, rst):

    off = "0" if rst[1] else "1"
    rstOff = f"#3\n\t{rst[0]} = {off};"

    return f"""// Create Date:  {date_time}
// Project Name: {moduleName}

`timescale {scale}

module {moduleName}_tb;
{paramsStr}
    //Creación de regs y wires
{regStr}
```

```
{wireStr}
    //Instanciar el top
    {moduleName} UUT(.*);

initial
  begin
    $dumpfile("{moduleName}_tb.vcd");
    $dumpvars (1, {moduleName}_tb);
{varInit}
    {rstOff if hasRst else ""}
{mainSequence}
    #4
    $finish;
   end
 {f"always forever #0.5 {clk} = ~{clk};" if hasClk else ""}
endmodule
    """
```

main.py

```
import sys
import re
from strFuncs import *
from displayMenu import *
from datetime import datetime

# Get the current time
current_time = datetime.now()
date_time = current_time.strftime("%m/%d/%Y, %H:%M:%S")

# Global regex and variables
re_com = r'\/\/[^\n]*|\/\*((?!\*\/).)*\*\/'

re_module_name = r'module\s+([_a-zA-Z]\w*)'

'''
1st group --> input|ouput|inout
3rd group --> logic | reg
4th group --> bus size
5th group --> variables separated by coma
'''
```

```
re_inout =
r'(input|output|inout)(\s+(reg|logic))?(\s*\[[^\]]+\]\s*|\s+)((?!input|
output|inout|reg|logic)[_a-zA-Z]\w*(,\s*(?!input|output|inout|reg|logic
)[_a-zA-Z]\w*)*)'

'''
1st group --> param_name
2nd group --> param_size
'''
re_parameters = r'((parameter)\s+(\w*)\s*\=\s*((\d+\'(b|h|d))?\w+))'

'''
key --> variable name
var_struct --> (name, size, type, funcType)
{
    key: var_struct
}
'''
# Global dictionaries
input_dicc = {}
output_dicc = {}
inout_dicc = {}


if __name__ == "__main__":

    # Override definition
    fOverride = False
    scaleOverride = False
    clkOverride = False
    rstOverride = False
    forOverride = False

    # Read args
    if len(sys.argv) <= 1:
        print("Missing arguments!")
        sys.exit(0)
    elif len(sys.argv) == 2:
        if sys.argv[1] == "--help":
            printHelp()
            sys.exit(0)
        inputFile = sys.argv[1]
```

```
    else:
      for i in range(1, len(sys.argv)):
          if sys.argv[i][0] == '-':
              if sys.argv[i] == "--help":
                  printHelp()
                  sys.exit(0)
              for j in range(1, len(sys.argv[i])):
                  if sys.argv[i][j] == 'r':
                      funcOverride = 'random'
                      fOverride = True


                  elif sys.argv[i][j] == 'a':
                      funcOverride = 'up'
                      fOverride = True
                  elif sys.argv[i][j] == 'd':
                      funcOverride = 'down'
                      fOverride = True
                  elif sys.argv[i][j] == 't':
                      scaleOverride = True
                  elif sys.argv[i][j] == 'c':
                      clkOverride = True
                  elif sys.argv[i][j] == 's':
                      rstOverride = True
                  elif sys.argv[i][j] == 'f':
                      forOverride = True
          else:
              inputFile = sys.argv[i]

  # Open the design file
  f = open(inputFile, 'r')
  textC = f.read()  # Read it as a string
  f.close()  # Close the file

  print("Welcome to the testbench generator!")

  # Clk name
  clk = "clk"
  if clkOverride:
      clk = getClk()
```

```
# (rst name, active HIGH?)
    rst = ("rst", True)
    if rstOverride:
        rst = getRst()

    # Remove comments from the file
    text = re.sub(re_com, "", textC, flags=re.DOTALL)

    # Get the module name
    moduleName = re.findall(re_module_name, text)[0]
    # Get all parameters
    params_list = re.findall(re_parameters, text)
    # Get all the inputs and outputs in the text
    inout_list = re.findall(re_inout, text)
    # Give proper format to parameters within the testbench
    # Iterate over parameters
    paramsStr = ""
    for par in params_list:
        paramsStr += "\n\t" + par[0] + ";"

    # Flags for clk and rst (verify if sequential or combinational
verilog design)
    hasClk = False
    hasRst = False

    # Iterate over the inputs and outputs
    for m in inout_list:
        '''
        Groups in the regex match
        1st group --> input|output|inout
        3rd group --> logic | reg
        4th group --> bus size
        5th group --> variables separated by coma
        var_struct --> (name, size, type, funcType)
        '''
        varList = m[4].split(",")  # Get variables list
        for var in varList:  # Iterate over the variables list

            # Create the tuple to save the variable info
            varTuple = [var.strip(), m[3].strip(), m[2], '']
            if not hasClk:
                hasClk = varTuple[0] == clk
```

```
            if not hasRst:
                hasRst = varTuple[0] == rst[0]

            # Save the tuple in the appropriate dictionary
            if(m[0] == "input"):
                if varTuple[0] != clk and varTuple[0] != rst[0]:
                    if fOverride:
                        varTuple[3] = funcOverride
                    else:
                        varTuple[3] = displayMenu(varTuple)
                input_dicc[varTuple[0]] = varTuple
            elif(m[0] == "output"):
                output_dicc[varTuple[0]] = varTuple
            else:  # inout
                inout_dicc[varTuple[0]] = varTuple
    # User type in iterator for simulation values
    forIt = 10
    if forOverride:
        forIt = selectForIterations()

    # User type in the timescale for the testbench
    scale = "1ns/1ps"
    if scaleOverride:
        scale = timescale()

    # Generate variable delcarations in SystemVerilog format
    regStr = generateInputTb(input_dicc, inout_dicc)
    wireStr = generateOutputTb(output_dicc)
    varInit = variableInit(input_dicc, clk, rst)
    mainSequence = generateMainSequence(
        input_dicc, forIt, clk, rst)

    # Open the test bench file in utf8 encoding
    tbName = inputFile[0:len(inputFile)-3]+"_tb.sv"
    tb = open(tbName, 'w', encoding='utf8')

    # Write the file with the string formatted appropriately
    tb.write(getTBString(date_time, moduleName, paramsStr, regStr,
                         wireStr, hasClk, hasRst, varInit,
mainSequence, scale, clk, rst))
    # Close the file
    tb.close()
    print("Done")
```

Conclusions

We present an auto-generated Testbench tool for Verilog that facilitates the elaboration of test benches for Verilog designs for further functional verification. As we can see, based on the example in Table 1, writing such Verilog testbenches could represent a time demanding task if manually input.

We try to think in all the different combinations of styles of programming in Verilog. For this, we use the tool regex to localize all the essential words. These are minor issues that can be solved with the complement of dictionaries.

The team managed to generate a collaborative synergy and we all got to learn new things and find better ways to code in Python.