Python Based Testbench Generator - Verilog File To Verilog Testbench

Mohit Agarwal K (19BEC073)
Dept. of Electronics and Communication Engineering
Institute of Technology,Nirma University
Ahmedabd,India
19bec073@nirmauni.ac.in

Himanshu Nakrani (19BEC075)
Dept. of Electronics and Communication Engineering
Institute of Technology,Nirma University
Ahmedabad, India
19bec075@nirmauni.ac.in

***Abstract*—In this paper we discussed about Verilog test-bench generation code that we created. This code takes a Verilog file as an input and gives back an auto-generated test-bench. Testbench is automatically written in the correct sequence first module and its name, ports, then UUT part, then the proper delay with the stimuli and then the module ends. Papers discussion are as in sequence of theory on Verification , describing the Testbench which is then followed by the advantages of testbench then methodology is done with a hands-on of SR latch future in the paper the code is explained starting from the creating an object, constructor to printing endmodule in the Verilog testbench.**

***Index Terms*—Python,Testbench,Testbench Generator,Verilog**

I. Verification

Design verification ensures that the design continues to comply with all system requirements throughout all levels of abstraction, from the system behavioural level to the block implementation level, and continuing through integration of the physical units. You will need to integrate, structure, and coordinate your verification methods to minimize the probability that the development team misses errors, especially complex system-level design errors, until much later in the design process. Such large-scale redesign is much more costly than iteration confined to a single design phase. The industry currently uses event-driven simulation as the most common design verification technology at the behavioural and the functional level. This technology is practical, well understood, and mature, but unable to keep up on its own with Moore's law: that design complexity grows at an exponential rate, approximately doubling every 12 to 18 months. Faster technologies are available, for example, hardware acceleration, cycle-based simulation and emulation, but are less applicable at the higher levels of abstraction early in the design process. The key to the design verification challenge is to not rely on technological advances alone, but to carefully plan and execute the overall design and design verification strategy. A viable design verification plan must include the following elements:

- Use of the appropriate technology for each phase of the design process This requires understanding the available technologies, the needs of the project, and the verification strategy. Depending upon the design phase, appropriate technologies can be anything from C-based stochastic analysis to a hardware prototype.
- Creation of appropriate, easily-selectable simulation configurations must be designed to focus verification efforts on key partitions of the design, while filling in the rest of the system context with behavioural models. The number and design of configurations also heavily depends upon the resources available, both of hardware and of personnel.
- Development of formal, well-structured testbench templates, the use of testbench templates facilitates test generation, maintenance, and modification. Test procedures, test data, and test timing should be separated, and good programming practices, such as parameterization, should be used.
- Generation of appropriate tests in a timely manner.

II. Test-bench

A simple testbench applies vectors to the design under test (DUT) and the user manually verifies the results. Non-trivial projects use some form of "intelligent" or "smart" testbench that reacts to the DUT, e.g., for a bus protocol that requires some kind of handshake mechanism. Such sophisticated testbenches are self-checking, i.e., they automatically verify the results. Sophisticated testbenches typically instantiate modules for stimulus generation and response checking under a top-level "wrapper", to more easily swap different tests and design configurations "in" and "out". You can write testbench code to dynamically react to keyboard or script input and to execute instructions from a file. The testbench code can be in Verilog or in the C programming language.

*1) **Advantages of Test-bench**:*

- Higher level of abstraction for creating test sequences and data.
- Sequence of tests and data can be easily changed by editing the external file.
- A text file is human readable and is easier to understand than in-line, looped, or arrayed stimulus.
- Tests and testbench are modular – additional tests can be easily added.

III. METHODOLOGY

In this section our approach to create a test-bench file has been discussed. every test-bench file has some similar structure. Using that structure we created different methods in python to and then using file handling in python we created a separate test-bench file, that we can later use for verification of our Verilog module.

A. *Implementation*

we can use this code from command-line by specifying name of input Verilog file and name of output test-bench that we want. that is shown in figure1.

Fig. 1. Running the script from command prompt

In 2 input Verilog file and corresponding output testbench file is shown that we get after running this code.

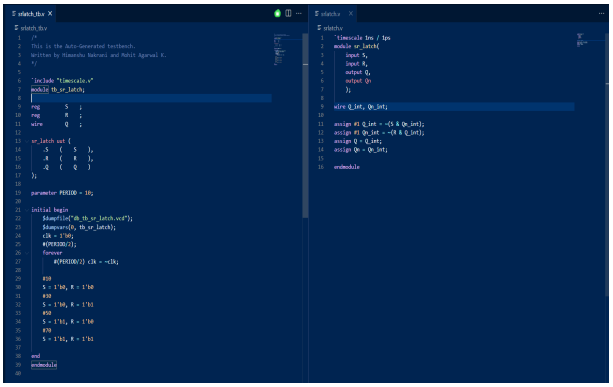

Fig. 2. input verilog file and output verilog testbench file

B. *Code Explanation*

This code takes input file name as command line arguments. If user haven't specified the verilog file name it will throw an error and stop execution of the program. And if user has given the file name, it will store it and pass it to the TestbenchGenerator object along with output file name.

then it creates 'tbgen' object, which is the instantiation of TestbenchGenerator class. In TestbechGenerator class we have many methods to print specific part of the test-bench. These methods prints the test-bench code into the output file.

figure 4 shows the constructor of TestbenchGenerator class it takes input and output file names as parameters. And then it initializes different fields of our 'tbgen' object. Like, name of

```python
if __name__ == "__main__":

    print('''------------------------------------------------------------------''')
    print('''---------------------- Generating the testbench.-------------------''')
    print('''------------------------------------------------------------------''')

    output_file_name = None

    if len(sys.argv) == 1:
        sys.stderr.write("ERROR: You have not specified a input file name.\n")
        print('''------------------------------------------------------------------''')
        print("HOW TO USE: testbench-generator input_verilog_file_name [output_testbench_file_name]")
        print('''------------------------------------------------------------------''')
        sys.exit(1)
    elif len(sys.argv) == 3:
        output_file_name = sys.argv[2]

    tbgen = TestbenchGenerator(sys.argv[1], output_file_name)

    tbgen.comment()
    tbgen.print_module_head()
    tbgen.print_wires()
    tbgen.print_dut()
    tbgen.print_clock_gen()
    tbgen.print_testbench()
    tbgen.print_module_end()

    tbgen.close()
```

Fig. 3. main method

```python
class TestbenchGenerator(object):

    def __init__(self, verilog_file_name=None, output_file_name=None):
        self.verilog_file_name = verilog_file_name
        self.verilog_file = None
        self.output_file_name = output_file_name
        if (output_file_name == None):
            self.output_file = sys.stdout
        self.vcont = ""
        self.mod_name = ""
        self.pin_list = []
        self.clock_name = 'clk'
        self.reset_name = 'rst'

        self.input_pin = []

        if verilog_file_name == None:
            sys.stderr.write("ERROR: You have not specified a input file name.\n")
            sys.exit(1)
        else:
            self.open()
        self.parser()
        self.open_outputfile()
```

Fig. 4. Constructor

Verilog filename, Verilog file handler, output file name, output file handler, module name, pin list, etc. Then it calls method open(), open_outputfile() and parser().

```python
def open(self, verilog_file_name=None):
    if verilog_file_name != None:
        self.verilog_file_name = verilog_file_name
    try:
        self.verilog_file = open(self.verilog_file_name, 'r')
        self.vcont = self.verilog_file.read()
    except (Exception, e):
        print("ERROR:Open and read file error.\n ERROR:    %s" % e)
        sys.exit(1)

def open_outputfile(self, output_file_name=None):
    try:
        if (output_file_name == None):
            if (self.output_file_name == None):
                ofname = "tb_%s.v" % self.mod_name
                self.output_file = open(ofname, 'w')
                print("You have not specify a output file name, use '%s' instead." % ofname)
            else:
                self.output_file = open(self.output_file_name, 'w')
                print('''----------------- Output file is '%s'. ----------------''' % self.output_file_name)
                print('''------------------------------------------------------------------''')
        else:
            self.output_file = open(output_file_name, 'w')
            print("Output file is '%s'." % output_file_name)
            print('''------------------------------------------------------------------''')
    except Exception as e:
        print("ERROR:open and write output file error. \n ERROR:    %s" % e)
        sys.exit(1)
```

Fig. 5. open() and open_outputfile() methods

Open method opens the Verilog file in read mode using verilog_file handler and reads the content of the file in 'vcont' variable. open_outputfile() method opens the output test-bench file in write mode. If the user has specified the file name it will use it to name the output file. If not, then if the user

```
def clean_other(self, text):
    text = re.sub(r"//[^\n^\r]*", '\n', text)
    text = re.sub(r"/\*.*\*/", '', text)
    text = re.sub(r"[^\n^\r]+`[^\n^\r]*", '\n', text)
    text = re.sub(r'    +', ' ', text)
    return text

def parser(self):
    print("------------------------ Parsing...------------------------ ")
    print('''--------------------------------------------------------------------''')
    mod_pattern = r"module[\s]+(\S+)[\s]+\([^\)]*\)[\s\S]*"

    module_result = re.findall(mod_pattern, self.clean_other(self.vcont))
    self.mod_name = module_result[0]

    self.parser_inoutput()
    self.find_clk_rst()
```

Fig. 6. parser() method

has specified a module name it will use it. Default name format is 'tb_module name'. Otherwise it will name the file by default name. Parser method removes all the commands and find the module name and stores it in mod_name variable. It uses regular expression available in python to clean all other things and find module name. Then it calls parser_inoutput() and find_clk_rst() method.

```
def parser_inoutput(self):
    pin_list = self.clean_other(self.vcont)

    comp_pin_list_pre = []
    for i in re.findall(r'(input|output|inout)[\s]+([^;,\)]+)[\s]*[;,]', pin_list):
        comp_pin_list_pre.append((i[0], re.sub(r'^reg[\s]*', "", i[1])))

    comp_pin_list = []
    type_name = ['reg', 'wire', 'wire', "ERROR"]
    for i in comp_pin_list_pre:
        x = re.split(r']', i[1])
        type = 0;
        if i[0] == 'input':
            type = 0
        elif i[0] == 'output':
            type = 1
        elif i[0] == 'inout':
            type = 2
        else:
            type = 3

        if len(x) == 2:
            x[1] = re.sub('[\s]*', '', x[1])
            comp_pin_list.append((i[0], x[1], x[0] + ']', type_name[type]))
        else:
            comp_pin_list.append((i[0], x[0], '', type_name[type]))

    self.pin_list = comp_pin_list
```

Fig. 7. parser_inoutput() method

```
def find_clk_rst(self):
    for pin in self.pin_list:
        if re.match(r'[\S]*(clk|clock)[\S]*', pin[1]):
            self.clock_name = pin[1]
            print("I think your clock signal is '%s'." % pin[1])
            print('''--------------------------------------------------------------------''')
            break

    for pin in self.pin_list:
        if re.match(r'rst|reset', pin[1]):
            self.reset_name = pin[1]
            print("I think your reset signal is '%s'." % pin[1])
            print('''--------------------------------------------------------------------''')
            break
```

Fig. 8. find_clk_rst() method

parser_inoutput() again using the regular expression to find input and output pins in the file. Is assigns list tuples to pin_list which contains of the type of the pin (input, output or inout), the name of the pin, and the type of the pin(reg, wire, or ERROR). find_clk_rst method finds if the clock and reset signal is available in our module or not. It searches for 'clk' or 'clock' and if found it assigns pin name to clock_name variable. Similarly it searches for 'rst' or 'reset' and if found it assigns pin name to reset_name variable.

```
def print_module_head(self):
    self.print_to_file("`include \"timescale.v\"\nmodule tb_%s;\n\n" % self.mod_name)

def print_module_end(self):
    self.print_to_file("endmodule\n")

def print_to_file(self, text):
    self.output_file.write(text)

def close(self):
    if self.verilog_file != None:
        self.verilog_file.close()
    print('''------------------------ Output finished. ------------------------''')
    print('''--------------------------------------------------------------------''')
```

Fig. 9. print_module_head(), print_module_end(), print_to_file() and close() methods

Every test-bench has same predefined module head part. print_module_head() method prints module head to test-bench file. And print_module_end() prints 'endmodule' at the end of test-bench file. print_to_file() method is created to write ant text directly into file. It takes text as an input parameter and then writes back text into test-bench file. Close() method closes all files when all work is done.

```
def align_print(self, content, indent):
    row_len = len(content)
    col_len = len(content[0])
    align_cont = [""] * row_len
    for i in range(col_len):
        col = [x[i] for x in content]
        max_len = max(list(map(len, col)))
        for i in range(row_len):
            l = len(col[i])
            align_cont[i] += "%s%s" % (col[i], (indent + max_len - l) * ' ')

    align_cont = [re.sub('[ ]+$', '', s) for s in align_cont]
    return "\n".join(align_cont) + "\n"
```

Fig. 10. align_print() method

align_print() method takes a string and returns the string where all the columns are aligned with specified number of spaces to indent the output.

```
def print_dut(self):
    max_len = 0
    for cpin_name in self.pin_list:
        pin_name = cpin_name[1]
        if len(pin_name) > max_len:
            max_len = len(pin_name)

    self.print_to_file("%s uut (\n" % self.mod_name)

    align_cont = self.align_print([("", "." + x[1], "(", x[1], '),') for x in self.pin_list], 4)
    align_cont = align_cont[:-2] + "\n"
    self.print_to_file(align_cont)

    self.print_to_file(");\n")

def print_wires(self):
    self.print_to_file(self.align_print([(x[3], x[2], x[1], ';') for x in self.pin_list], 4))
    self.print_to_file("\n")
```

Fig. 11. print_DUT() and print_wire_reg() methods

The test bench applies stimulus to the DUT. To do this the DUT must be instantiated in the test bench, print_dut() method takes care of it. print_wire_reg() method declares 'reg' and 'wires' in the test-bench.

print_clock_gen() method writes the generation of clock signal at the specified period in the test-bench.

Then print_testbench() method is heart of out test-bench generator program. It will write testbench in the output file.

```
def print_dut(self):
    max_len = 0
    for cpin_name in self.pin_list:
        pin_name = cpin_name[1]
        if len(pin_name) > max_len:
            max_len = len(pin_name)

    self.print_to_file("%s uut (\n" % self.mod_name)

    align_cont = self.align_print([("", "." + x[1], "(", x[1], '),') for x in self.pin_list], 4)
    align_cont = align_cont[:-2] + "\n"
    self.print_to_file(align_cont)

    self.print_to_file(");\n")

def print_wires(self):
    self.print_to_file(self.align_print([(x[3], x[2], x[1], ';') for x in self.pin_list], 4))
    self.print_to_file("\n")
```

Fig. 12. print_clock_gen() method

First of all using pin_list we have determined which are input pins in out dut. Then we based on number of input pins we have written logic for testbench generation. If there are less number of input pins then we can check all the combination at inputs at specific delay. But as number of input pin increases it become very inconvenient and takes so much time. Instead we have used the concept of randomization. We have fixed how many the number of times testcases will be generated and then using python's random module we randomly applied inputs to make testcases. And printed this all into test-bench file.

```
def print_testbench(self):
    for pin in self.pin_list:
        input_pin = []
        for pin in self.pin_list:
            if 'input' in pin:
                if (pin[1] not in input_pin):
                    if not (re.match(r'^[\S]*(clk|clock)[\S]*', pin[1])) and not (re.match(r'rst|reset', pin[1])):
                        input_pin.append(pin[1])
    input_no = len(input_pin)
    input = [0, 1]
    delay = 10

    if input_no == 1:
        for i in ["1'b0", "1'b1"]:
            self.print_to_file("\t#{} \n".format(delay))
            delay = delay + 20
            self.print_to_file("\t")
            self.print_to_file("{0} = {1}".format(input_pin[0], i))
            self.print_to_file("\n")
```

Fig. 13. testbench() method with logic for one input pin

```
    elif input_no == 6:
        for i in range(10):
            i = random.choice(input)
            j = random.choice(input)
            k = random.choice(input)
            l = random.choice(input)
            m = random.choice(input)
            n = random.choice(input)
            self.print_to_file("\t#{} \n".format(delay))
            delay = delay + 20
            self.print_to_file("\t")
            self.print_to_file(
                "{0} = 1'b{1}, {2} = 1'b{3}, {4} = 1'b{5}, {6} = 1'b{7}, {8} = 1'b{9}, {10} = 1'b{11}".format(
                    input_pin[0], i, input_pin[1], j, input_pin[2], k, input_pin[3], l, input_pin[4], m,
                    input_pin[5], n))
            self.print_to_file("\n")
```

Fig. 14. randomization concept

IV. Conclusion

In this paper we have implemented python code to generate testbench automatically once the Verilog code file is available for the same. It is very generalized auto-generated Verilog testbench so there are few limitations or few future scope of this paper are decreasing the generalized code by separating combinational and sequential code, adding different or random delays in the testbench and many more scopes are there.

Acknowledgment

This paper was performed under the guidance of proff. Vaishali Dhare and Proff. Usha Mehta of Electronics and Communication Department,Institute of Technology Nirma University. The authors of the paper are thankful to the university and the professors for the guidance in completion of the term paper as a part of the teaching and learning process.

References

[1] M. Nodine, "Automatic Testbench Generation for Rearchitected Designs," 2007 Eighth International Workshop on Microprocessor Test and Verification, 2007, pp. 128-136, doi: 10.1109/MTV.2007.11.

[2] D. Hunter, "Some lessons learned on constructing an automated testbench for evolvable hardware experiments," Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753), 2004, pp. 1808-1812 Vol.2, doi: 10.1109/CEC.2004.1331115.

[3] F. Corno, M. Sonza Reorda, G. Squillero, A. Manzone and A. Pincetti, "Automatic test bench generation for validation of RT-level descriptions: an industrial experience," Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537), 2000, pp. 385-389, doi: 10.1109/DATE.2000.840300.

[4] Silva, Karina Melcher, Elmar Araujo, Guido Pimenta, Valdiney. (2004). An Automatic Testbench Generation Tool for a SystemC Functional Verification Methodology. 66-70. 10.1145/1016568.1016592.

[5] Srivastava, R., Gupta, G., Patankar, S., Mudgil, N. (2013). Automatic Test Bench Generation and Connection in Modern Verification Environments: Methodology and Tool. In: Gaur, M.S., Zwolinski, M., Laxmi, V., Boolchandani, D., Sing, V., Sing, A.D. (eds) VLSI Design and Test. Communications in Computer and Information Science, vol 382. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-42024-5_34

[6] K. R. G. da Silva, E. U. K. Melcher and G. Araujo, "An automatic testbench generation tool for a systemC functional verification methodology," Proceedings. SBCCI 2004. 17th Symposium on Integrated Circuits and Systems Design (IEEE Cat. No.04TH8784), 2004, pp. 66-70, doi: 10.1109/SBCCI.2004.241019.