Tree Data Framework for Code Generation: Application of Generating UVM Testbench for Complex Designs

Chenhui Huang
chuang@tenstorrent.com

Yu Sun
ysun@tenstorrent.com

Divyang Agrawal
dagrawal@tenstorrent.com

Tenstorrent Inc.
7717 Southwest Parkway, Bldg 3
Austin, TX 78735-9007

***Abstract*-Building and maintaining a testbench is expensive. Especially for the complex design such as CPU which is highly customizable such that complexity can be pushed into the microarchitecture and implementation. This paper presents a novel framework called Gingko to auto-generate testbench environments that eliminates a big class of errors, increases code consistency and reduces overall design time. This framework is based on a "tree data structure" which describes the various testbench components and their dependencies as nodes on a tree. The implementation requires the handler code for each tree node to be invoked at generation time. With a pre-described dependency, it then stitches together various components automatically. Using the Gingko code generation framework, we successfully developed a UVM testbench auto-generation flow for various sub-units of a RISC-V CPU project. The generated testbenches not only include the basic structural code and signal connection but also contain a variety of utility functions such as sequence library generation, transaction debug tracker and transaction replay. Using Gingko to generate UVM testbenches allowed different teams to have a consistent code quality and significantly sped up the bring-up process.**

I. Introduction

Code generation has been widely used in developing design verification components such as testbench [1]. Generating testbenches for complex hardware designs eliminates a large class of errors, increases code consistency and reduces overall design time [2]. The efficacy increases significantly when specifications remain a moving target well into the design phase compared with the traditional "hand-written" and "hand-maintained" methods. This paper proposes a novel framework called Gingko for the testbench generation using a "tree data structure".

Traditional testbench generation methods rely on code templates or code macro, where the design-related metadata is filled and the code is generated dynamically according to the desired scenario [3]. Unfortunately, as designs and the corresponding verification collateral becomes more complex, static generation approaches show less flexibility. In addition, managing and maintaining the template library with metadata becomes increasingly difficult [4]. A code generation framework based on the tree data structure is inherently more adaptable. Using the framework of Gingko, we build a reliable flow to generate the Universal Verification Methodology (UVM) testbenches for complex designs including different sub-units of CPU. Under the framework of Gingko, the testbench components are represented and stored within the tree data structure, which is used to concisely and reliably describe their hierarchical and dependency information. In the tree structure, different testbench components are symbolized as tree nodes with the parent-and-child relationship. Each tree node under the framework has its own Component Element (CE) type associated with a callback handler code which can be used to construct the code to generate with the information in the CE and its dependent CEs. For example, a tree node with a CE type of UVM Env contains the handler code that connects dependent sub-layer agents using systemverilog. The handler code of nodes in the tree structure can be used to generate the whole testbench when the node is called during the code-generating phase. During the testbench code-generating phase, the post-order traversal algorithm is used to visit all the nodes in the tree and trigger the callback handler code in that traversal order. Following that sequence, the testbench code is generated with all the components created in the desired order.

This automation framework also includes many utilities that help users efficiently and quickly build the tree data structure per the design specification. They are coded and managed by a Python library called ezGingko, which consists of a design auto extraction utility, integrated MySQL database and web application graphic user interfaces (web gui). The utilities are built using Python with open-source libraries. The design auto extraction utility uses FSDB debug and extract tools to obtain the design information including RTL signal name, signal type, signal width and

signal hierarchy directly from the design instead of restoring from any metadata or parsing the RTL design files. These are then uploaded onto the database. The user can then pull the information in the database to customize and construct the tree structure that describes the testbench components. In addition, the hierarchy can be set up using the web gui or via a script.

Using Gingko as the framework, we successfully developed a flow to generate UVM testbenches for various sub-units of a high-performance CPU. The generated testbenches not only include the basic structural code and signal connection, but also contain a variety of utility functions such as sequence library generation, transaction debug tracker, transaction replay, etc. Using this flow to generate UVM testbenches allowed different teams to have a consistent code quality and significantly sped up the bring-up process.

The rest of this paper is organized as follows:

Section 2 will give an overview of the phases and procedures of Gingko to generate testbench code from the design and what Gingko code generation framework will produce.

Section 3 will describe in detail how the tree data structure is used in the Gingko code generation framework.

Section 4 will demonstrate an example of using Gingko to generate a UVM testbench for a simple Arithmetic-Logic Unit (ALU).

Section 5 will discuss the usage of SQL databases in the framework and the benefits the database can bring.

Section 6 will exhibit the usage of open-source packages in the code generation framework.

Section 7 will be a discussion about the strength of Gingko over the other existing approach of testbench generation methodology. This section will also cover the limitations and challenges of the Gingko framework.

Section 8 will be the conclusion and potential future work for the Gingko framework.

II. Procedures of Testbench Generation from RTL Design

The whole code generation framework consists of three phases: the **collecting phase**, the **configuration phase** and the **generating phase**. All these phases are managed and executed by a Python library called ezGingko. In this section, these phases will be explained in detail.

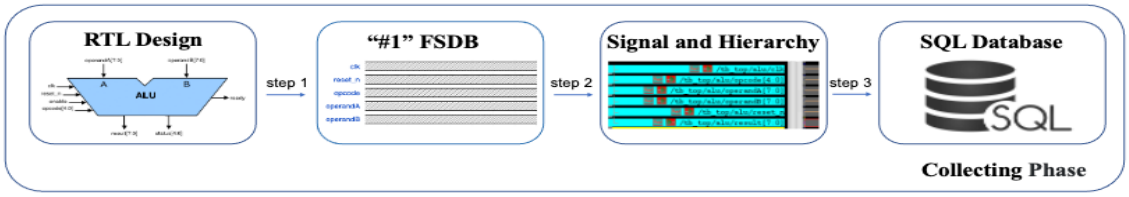

Figure 1. Flow chart of the steps in the Collecting Phase of the Gingko code generation framework

Figure 1 shows the flow chart of the steps in the **Collecting phase**. In the **Collecting phase**, we need an automatic way to extract and collect the RTL signals and hierarchical information from the design. In **step 1** of the phase, the RTL design will be run without any stimulus for 1 unit of a timestamp to generate the FSDB. By doing that, the signal names and hierarchical information will be stored in a universal format without parsing the RTL design files. In **step 2**, the Verdi fsdbextract and fsdbdebug commands will be used to extract the required signal and hierarchy information into text format and then be uploaded and stored in the SQL database in **step 3**. How the design information is stored efficiently in the SQL database will be discussed in detail in section 5 of the paper.

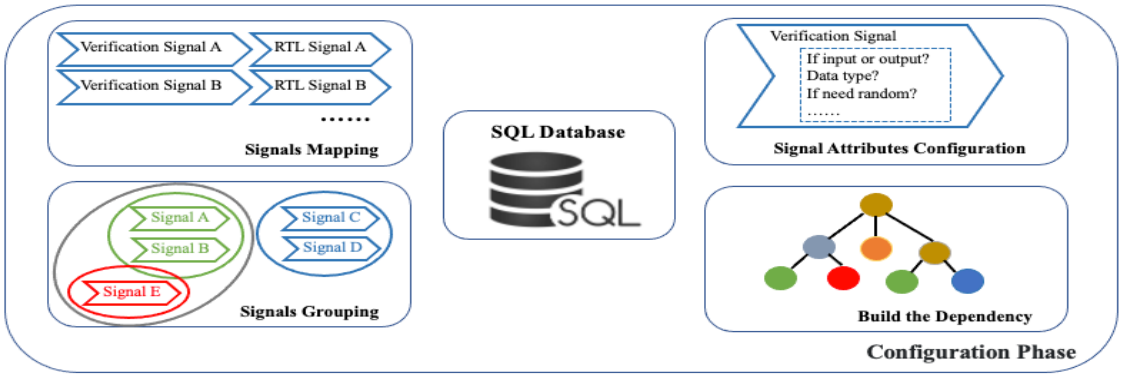

Figure 2. Different procedures in the Configuration Phase of the Gingko code generation framework

Figure 2 shows the four main procedures in the **Configuration phase** after the design information has been extracted and stored in the SQL database. In this phase, we will decide the structure and functionality of the testbench to be generated. The **Signals Mapping** procedure creates the maps between the verification signals and the RTL signals. The verification signals will be used in the interface components to connect the RTL signals in the testbench to be generated. The verification signal name will default to be the same as the RTL signal name. However, in this procedure, the user has an opportunity to configure the verification signal name if the RTL signal name is not concise to use. In the procedure of **Signal Attributes Configuration**, the user needs to configure the main attributes of verification signals such as signal data type and signal direction. These attributes will be mainly used to define the sequence items and control signals in the testbench. Then in the **Signals Grouping** procedure, the signals will be grouped based on the different functionalities used in the testbench as a CE. The CEs generated in this procedure will be the essential elements used to construct the testbench in the procedure of **Building the Dependency**. In this procedure, the dependency means the internal relationship between different testbench components. For example, in a typical UVM testbench, a UVM sequencer depends on a UVM sequence item. Since the sequencer can be defined and created only after defining a sequence item. This dependency can be more adaptably and accessibly described using a tree data structure. Therefore, we use the tree data structure to define the testbench components relationship, which will become the testbench structure. The tree data structure in the code generation framework will be discussed in section 3 of the paper.

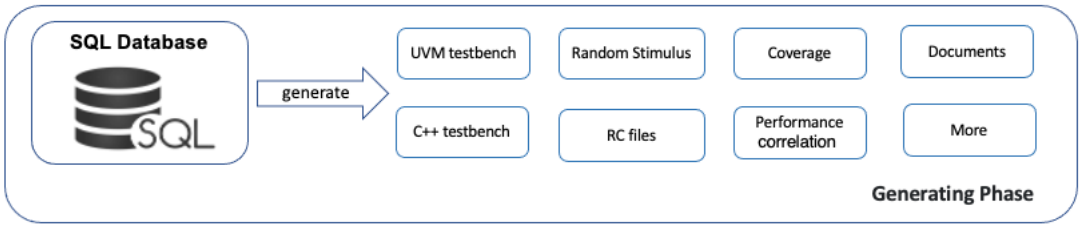

Figure 3. Generating Phase of the Gingko code generation framework

Figure 3 shows the **Generating Phase**, which generates the production of testbench code from the SQL database. Since all the testbench structure and signal information has been generated and stored in the SQL database, we can generate more than a UVM testbench. For example, based on different tree structure handler codes, we can generate a C++ testbench based on the same testbench logic structure and dependency. Using the signal information in the database, we can also generate the facilities for debugging, stimulus and coverage. Furthermore, we can use the framework to generate the documents and records of the design and test plan.

III. TREE DATA STRUCTURE FOR CODE GENERATION

The tree data structure is a widely used abstract data type representing and manipulating hierarchy and dependency information [5]. A tree data structure can inherently represent the testbench components and structure. For example, Figure 4 shows a typical UVM agent and its dependent testbench components represented in a tree structure. The agent node is the parent node of the driver, sequencer and monitor. Since the driver, sequencer and monitor are instantiated and used in the class of agent, the tree data structure can be used to describe the dependency concisely. After the testbench structure is created in a tree data structure, the post-order traversal algorithm visits all the nodes to create testbench components in the desired order.

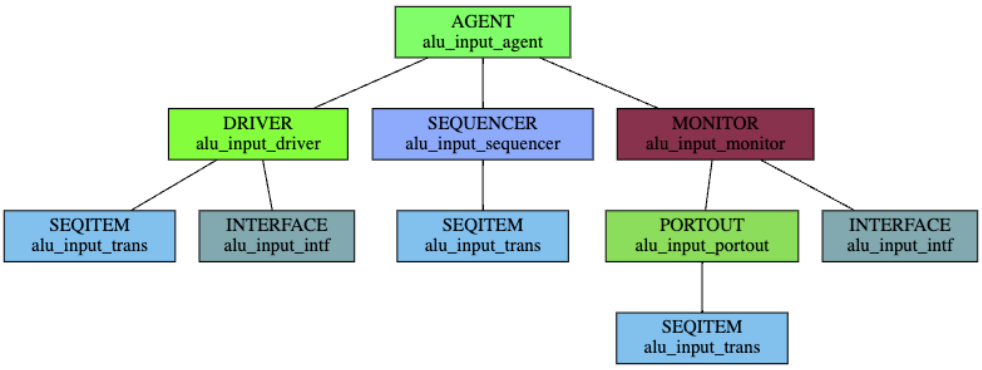

Figure 4. Example of using tree data structure to represent typical testbench components generated by the web gui of Gingko

Each node in the testbench tree has a Component Element (CE) type representing the functionality of the testbench component. For example, Figure 4 shows different testbench components with different CE types using different colors. The node alu_input_trans has a CE type called SEQITEM, which means that the node will become a UVM sequence item in code generation. Each CE type is associated with a piece of Python callback handler code. During the post-order traversal algorithm to access each node of the testbench tree, the handler code will be called using the node's and its child nodes' information to generate the desired testbench code. Figure 5 shows the piece of sample Python handler code for the CE type SEQITEM on the left side. When the handler code is called during the code-generating phase, it will generate the code for an actual UVM sequence item with the information from the database.

Handler code for SEQITEM

```
out_stream.write('''

time _time_;
local int _uid_;

function int uid();
  return _uid_;
endfunction

`uvm_object_utils_begin(%s)
  `uvm_field_int(_uid_, UVM_DEFAULT)''' % instance_type)
      self.add_uvm_field_macros(tree_iter = member_data.
        tree_iter, out_stream=out_stream, prefix="    ")
      out_stream.write('''
`uvm_object_utils_end

function new(string name = "unnamed-''' + instance_type + '''
  ");
  super.new(name);
  _time_ = $time; //feel free to update this timestamp
  _uid_ = seq_item_tracker::next_id();''')
      self.instantiate_members(tree_iter=member_data.
        tree_iter, out_stream=out_stream)
      out_stream.write('''
endfunction
```

Generated code for SEQITEM

```
time _time_;
local int _uid_;

function int uid();
  return _uid_;
endfunction

`uvm_object_utils_begin(alu_input_trans)
  `uvm_field_int(_uid_, UVM_DEFAULT)
  `uvm_field_int(val_operandA, UVM_DEFAULT)
  `uvm_field_int(val_operandB, UVM_DEFAULT)
  `uvm_field_int(val_opcode, UVM_DEFAULT)
`uvm_object_utils_end

function new(string name = "unnamed-alu_input_trans");
  super.new(name);
  _time_ = $time; //feel free to update this timestamp
  _uid_ = seq_item_tracker::next_id();
  //No need to instantiate "SIGNAL_MAP" "val_operandA"
  //No need to instantiate "SIGNAL_MAP" "val_operandB"
  //No need to instantiate "SIGNAL_MAP" "val_opcode"
endfunction
```

Figure 5. Sample of handler code for the CE type SEQITEM and the corresponding generated UVM code

The tree data structure gives the user a more flexible and concise way to describe the testbench structure and the dependency between different testbench components. More importantly, it makes the user describe the testbench components, connections and interactions as independent of the targeting language and separates the testbench structure from the specific programming language or verification methodology. By calling different flavors of handler code, it is almost effortless to generate an entirely different testbench with different languages while the testbench structure keeps the same.

IV. Demo of Generating a UVM Testbench for a Simple ALU

In this section, we will demonstrate an example using Gingko to generate a UVM testbench for a simple ALU design by the procedures discussed in section 2 of the paper. Figure 6 shows a typical ALU design's input and output signals specification. The design has five parallel buses consisting of one input opcode, two input operands, one result output and one status output. In addition, there are several control signals including enable and ready signals. We will generate a self-checking testbench to simulate the RTL and test the ALU's functionality by the Gingko code generation framework.

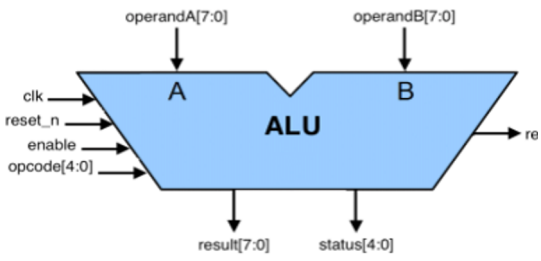

Figure 6. A simple ALU block diagram with input and output signals definition

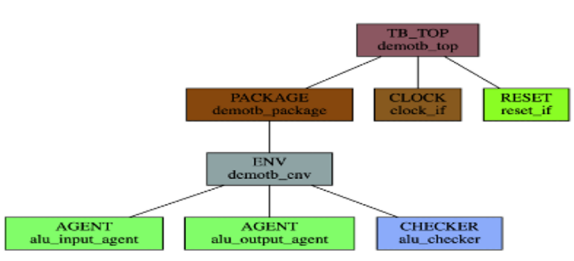

Figure 7. The testbench tree structure for a simple ALU generated by the web gui of Gingko

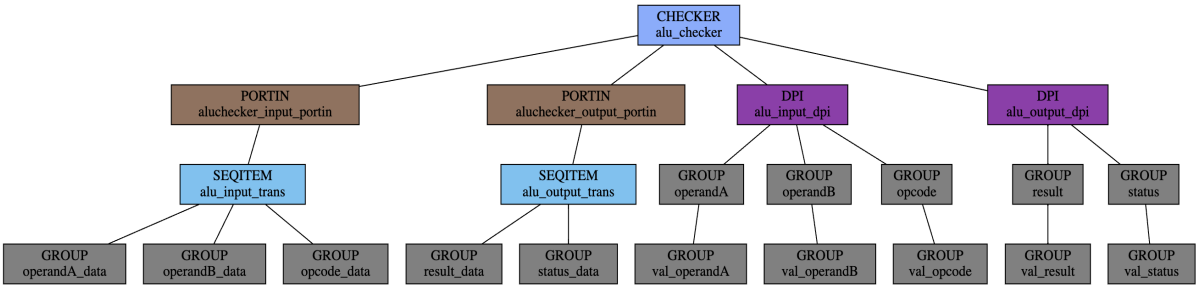

Figure 8. The tree structure of ALU checker (scoreboard) structure generated by the web gui of the code generation framework

Figure 7 shows the proposed testbench structure for the ALU from the top using the tree data structure generated by the web gui of the Gingko framework. In the testbench structure, there is an active agent, a passive agent and a checker (scoreboard) within the testbench environment. The active agent (alu_input_agent) contains the sequencer, driver and monitor, which is responsible for the input stimulus of the testbench. Figure 4 demonstrates the detailed tree structure of alu_input_agent. The passive agent (alu_output_agent) contains the monitor to get the result and status of the ALU. The checker contains the ports connecting the monitors of both agents and two Direct Programming Interface (DPI) modules connecting to the golden reference model. The detailed tree structure of the checker is illustrated in Figure 8.

Using the procedures discussed in section 2 of the paper and the proposed testbench structure in this section, we can generate a self-checking UVM testbench from the ALU RTL design. Starting with the **collecting phase**, the ALU RTL will be run without any stimulus for 1 unit of a timestamp to extract the RTL signal names and hierarchical information from the design. The information will be uploaded and stored in the SQL database for the later phase to consume. Figure 9 shows the sample of the RTL signal names and hierarchical information stored in the SQL database. The detail of data storage in the database will be discussed in section 5 of the paper.

vcd_type	type	hier_level	hier	signal_hier	signal_range	signal_name	signal_level	bit_range	l	r	num_bits	_rtl_signal_sha256
reg	output	3	tb_top.alu		result[7:0]	result	1	7:0	7	0	8	8907ead2e400f09500
reg	output	3	tb_top.alu		status[4:0]	status	1	4:0	4	0	5	69583eb16563277c4(
reg	output	3	tb_top.alu		ready	ready	1		0	0	1	f11523034d577d60eae
wire	input	3	tb_top.alu		clk	clk	1		0	0	1	df1ff84801802fc0300
wire	input	3	tb_top.alu		reset_n	reset_n	1		0	0	1	0b04c011e3cc89e548
wire	input	3	tb_top.alu		operandA[7:0]	operandA	1	7:0	7	0	8	0eb5653a593ccaee6b
wire	input	3	tb_top.alu		operandB[7:0]	operandB	1	7:0	7	0	8	1b71504aad5d0d6279
wire	input	3	tb_top.alu		opcode[4:0]	opcode	1	4:0	4	0	5	1fcc909a0321f4511b73
wire	input	3	tb_top.alu		enable	enable	1		0	0	1	8b1dd81265c6ff97077

Figure 9. Sample of RTL signal names and hierarchical information stored in the SQL database

We execute four procedures to construct and configure the testbench using the tree data structure in the **configuration phase**. Therefore, we developed a library called ezGingko to help users quickly and easily configure the procedures using Python. Figure 10 and Figure 11 demonstrate the example code configuring the testbench for the ALU design. In the example code of Figure 10, we define the necessary signal attributes for the ALU opcode bus in a Python dictionary. In this dictionary, we also define the mapping relationship from the RTL opcode signal to the verification signal called val_opcode. The dictionary in Figure 10 is the fundamental bottom-level component under the Gingko framework, which is called a component element (CE). The CE will be later grouped with other CEs to become a higher-level CE according to their proposed functionality. For example, the opcode's CE and the operand's CEs are grouped to become the CE of the input sequence item. Then, the CE of the input sequence item will be grouped with the interface CE to become the driver CE. Later, we group the driver CE, sequencer CE and monitor CE to generate a typical active agent CE used for providing the stimulus to the ALU testbench. The dependency of the different CEs from bottom to up for the agent can be illustrated using the tree structure in Figure 4 of the paper. To make it easy, the ezGingko library provides a predefined class called Naive_TB_Agent to construct a typical agent like that. After defining the CEs of the two agents and the checker, we repeat the same grouping process to generate the higher-level components until the testbench top, where the sample code is shown in Figure 11. By using the class and functions in the ezGingko library, we can construct and configure the testbench components and their dependency briefly and effortlessly as a tree data structure under the framework of Gingko.

```
opcode = {
    "itemType":"signal",
    "signal":"opcode[4:0]",
    "atb_hier_define_name":"tb_top.alu",
    "seq":1,
    "type_hint":"input",
    "rand_mode":True,
    "sv_type":"logic",
    "map_to":"val_opcode",
    "invert":False,
    "l":-1,
    "r":-1,
    "pipe_line":1,
    "sig_function":"DATA",
    "allow_replace":False,
    "groupName":"opcode",
    "groupType":"GROUP"
}
```

Figure 10. Sample code used to do signal mapping and signal attributes configuration for the opcode bus

```
alu_input_agentIns = Naive_TB_Agent("alu_input",
                                    [operandA, operandB, opcode],
                                    [enable],
                                    [tb_clock, tb_reset_n],
                                    active = True
                                    )
tbCfg.addGrp(alu_input_agentIns)

demotb_env = Group("demotb_env", "ENV", [alu_input_agentIns.agent(),
  alu_output_agentIns.agent(), alu_checker])
tbCfg.addGrp(demotb_env)

demotb_package = Group("demotb_package", "PACKAGE", [demotb_env])
tbCfg.addGrp(demotb_package)

demotb_top = Group("demotb_top", "TB_TOP", [demotb_package, clock_if, reset_i
tbCfg.addGrp(demotb_top)
```

Figure 11. Sample code used to do the signal grouping and dependency building for the ALU testbench

V. USING SQL DATABASE

We use the SQL database to store the testbench details as well as the design information in an efficient way. Using the database allows users to share data from team to team easily. More importantly, we are applying the Model-View-Controller (MVC) [6] pattern in this framework by using the database to separate data, logic and view instead of the traditional monolithic script to the code generation framework, which will be discussed more in section 6 of the paper. Another potential benefit of using the database is that different designs and testbench flavors will be remembered in a platform after more users with different backgrounds use the Gingko framework. It will provide a foundation and the possibility to use artificial intelligence to generate the testbench by using the machine learning algorithm to execute the procedures in the **configuration phase** discussed in this paper.

_signal_range	_rtl_signal_sha256	seq	type_hint	rand_mode	sv_type	map_to	invert	l	r	pipe_line	sig_function	_tb_signal_sha256
clk	df1ff84801802fc0300990d48c96481243e4abf0192bb630a32702b908e5ba2b	1	input	☑	logic	tb_clk	☐	-1	-1	0	CLK	0639bd82b7412e04e162
reset_n	0b04c011e3cc89e5484c1b374695873b868619dea311a8dfd8326bb62a843225	1	input	☑	logic	tb_reset	☐	-1	-1	0	RST	b72ec81da33d3052dcec
enable	8b1dd81265c6ff9707763cdc6c09472e820913a6bd2b9ee5bb9d5e5a27410c73	1	input	☑	logic	val_enable	☐	-1	-1	0	DATA	3d0567e3091a5cee288b
operandA[7:0]	0eb5653a593ccaee6be033284659662deda42e78393a95592249eaba2e57af0c	1	input	☑	logic	val_operandA	☐	-1	-1	0	DATA	c376156f8d8ccaebb007
operandB[7:0]	1b71504aad5d0d62793ee495b9de981e590d0744eafdd5e9540386a4850381e5	1	input	☑	logic	val_operandB	☐	-1	-1	0	DATA	4205814d548c709e1fb0
opcode[4:0]	1fcc909a0321f4511b737ad0a5fe6fc61e8c9754dba1637ab954a2d1d80c76a1	1	input	☑	logic	val_opcode	☐	-1	-1	1	DATA	c96ff7ad0b1e1ed86072b
ready	f11523034d577d60eae9914223584db9fc2e9ea6c1032e2bc4b49545d130f050	1	output	☑	logic	val_ready	☐	-1	-1	0	DATA	3da1f3b8b44395800612
result[7:0]	8907ead2e400f095007267e7d9fb9fcaccf8a86ae0420da198fc0b737d0bea30	1	output	☑	logic	val_result	☐	-1	-1	0	DATA	de237fc4cb235b8cdfd9
status[4:0]	69583eb16563277c4085b4a0a02a808d25181b86e1b9683d5121c050e8228101	1	output	☑	logic	val_status	☐	-1	-1	0	DATA	3045bd1c059e51e32daf

Figure 12. Sample of signal mapping data stored in the SQL database

Figure 9 and Figure 12 show the sample of data for the design and testbench information in the SQL database, respectively. Each entry of the SQL database table is associated with a SHA256 number, which is unique and created using the signal name and hierarchical information. The SHA256 number is also used to build the dependency connection among the testbench component elements as a tree data structure. By doing that, the dependency of the testbench components within the tree structure is stored and used efficiently as a tabular format.

Under the Gingko code generation framework, we have a web gui developed based on the Python Django package that lets users easily view and modify the data in the SQL database if necessary. By using the web gui to access the database, the user without any knowledge of Python can also construct and configure a testbench with a tree structure under the framework of Gingko.

Using SQL database as storage also reduces the development cycle of the whole framework since the read and write API for the data storage is a standard and commonly used domain-specific language SQL. We don't need to develop an extra API to access the data collection and the user does not have extra learning costs either.

VI. USAGE OF OPEN-SOURCE PACKAGES

We have used various open-source packages in the Gingko framework to reduce the development cycles and avoid duplicate work. Table I lists all the open-source packages used in the Gingko development.

TABLE I
OPEN-SOURCE PACKAGES USED IN GINGKO

Package Name	Usage in the Gingko Framework
Django	The framework for the web gui development
Graphviz	Used for plotting the tree structure for the testbench in the web gui
Pandas	Numerical table and data manipulation
SQLAlchemy	SQL toolkit and object-relational mapper for the Python
Treelib	An efficient implementation of tree data structure and traversal function in Python

VII. DISCUSSION

Gingko is a novel framework to auto-generate the testbench environments based on the tree data structure. The core idea of the framework is that it applies the concept of the Model-View-Controller (MVC) pattern to separate the Data, View and Logic instead of traditional monolithic script. In the framework, the Model is the tree data structure with the design and verification information stored in each tree node as the attributes. The Logic (Controller) is the different kinds of callback handler code associated with the group type of the tree nodes. The View is the generated testbench code with specific verification language and methodology. Using the MVC pattern for Gingko framework, testbench components, connections and interactions are described as independent of any targeting language and methodology, which is entirely different from traditional code-template-based or code-macro-based testbench generation methodology. Compared with the traditional approach, Gingko has the following strength.

1) Gingko has a more flexible way of describing the testbench structure using the tree data structure, which can be applied to all kinds of designs. Users can add or remove groups for any necessary testbench components based on the requirement by adjusting the tree structure. For example, the traditional UVM testbench has the concept of a UVM Agent consisting of all the subcomponents as a modular unit for code reuse purposes within the UVM Environment. In Gingko framework, we can easily remove that agent layer and group the components directly under the Environment in many applications. Since it is an auto-generated testbench, the user does not need to consider code reuse. Furthermore, the user can use the framework to generate more than testbench code and use the framework to generate the utilities such as coverage and debug facilities.

2) The user does not need to maintain many design metadata and template libraries. In the Gingko framework, the design information is directly extracted from the FSDB. Only a minimal amount of work is required to regenerate an entirely new testbench when the design changes during the verification execution phase.

3) Gingko uses the SQL database as storage for the information related to design and verification. By doing that, users can easily share the data from team to team using the database interface. More importantly, different designs and testbench flavors will be remembered in a platform after more users with different backgrounds use the Gingko framework. It creates a foundation and makes it possible for future development using artificial intelligence to completely auto-generate the testbench, stimulus and coverage from the design.

4) The main facility of Gingko is based on Python and SQL database, which is generally used in the industry. The user will spend less learning cost than learning new syntax in most template-based code generation methodologies.

There are some limitations and challenges for the Gingko testbench generation framework.

1) The existing code generation framework cannot provide meaningful and complicated sequences except for completely random stimulus. To generate a more meaningful sequence for the complicated scenario, the generated testbench still requires a lot of user's input and configuration.

2) For now, Gingko is still a proprietary tool with limited contribution to the framework. Therefore, there will be inadequacy in the diversity of handler code for different CE types associated with the tree nodes.

3) The Gingko framework requires uploading the design information to the SQL database. There will be a confidential issue after the framework is released into a public platform for different companies to use. A reliable authentication solution is required to keep the confidentiality of IP while the model of the framework can still learn the flavor of different designs and testbench.

VIII. Conclusion and Future work

We present a novel approach based on tree data structure to auto-generate testbench environments called Gingko that eliminates a significant class of errors, increases code consistency and reduces overall design time. Using Gingko, we successfully developed a flow to generate UVM testbenches for various sub-units of a RISC-V CPU project. This allowed different teams to have a consistent code quality and significantly sped up the bring-up process.

In the future, we will make more efforts to open-source the Gingko and make it a platform for more teams with different backgrounds to use and develop. In that case, the handler code for different CE types will be enhanced and expanded. Different flavors of testbench will be generated using the framework of Gingko. We will also use the facilities of Portable Test and Stimulus Standard (PSS) [7] to enhance the sequence model and diversify the stimulus to generate more meaningful sequences for the testbench. Meanwhile, we will attempt to apply the machine learning algorithms to the platform to replace the human efforts in the Configuration Phase of Gingko. For example, we may apply an unsupervised clustering algorithm for the CE grouping procedures and neural network to auto-configure attributes for each testbench component. We hope a comprehensive testbench with meaningful stimulus, complete structure and functional debug facilities can be generated using Gingko, which will reduce the execution cycles and efforts to bring up the silicon.

Acknowledgment

The authors would like to thank the people who encourage Gingko's development and many others who are using Gingko and provide helpful feedback including Srikanth Arekapudi, Yan Feng, Jon Wang, Abhijeet Badrike, Pratyush Jain and Colby Renfro. Also, great thanks to Dr. Xin He for his feedback on the paper.

References

[1] Kim Namdo, Young-Nam Yun, Young-Rae Cho, Jay B. Kim, and Byeong Min. "How to automate millions lines of top-level UVM testbench and handle huge register classes." *2012 International SoC Design Conference (ISOCC),* pp. 405-407. IEEE, 2012.

[2] P. Munier, "Challenges for Auto Code Generation and Verification," *2006 2nd IEEE Conference on Automotive Electronics*, 2006, pp. 19-20.

[3] Daeseo Cha, et al. "Metadata Based Testbench Generation Automation." *DVCON*, 2022.

[4] Lubars, Mitchell D. "Code reusability in the large versus code reusability in the small." ACM SIGSOFT Software Engineering Notes 11.1 (1986): 21-28.

[5] Donald Knuth (1997). "Section 2.3: Trees". The Art of Computer Programming. Vol. 1: Fundamental Algorithms (Third ed.). Addison-Wesley. p. 308.

[6] Reenskaug, T. M. H. (1979). The original MVC reports.

[7] "Portable Test and Stimulus Standard Version 2.0" [online]. Available: https://www.accellera.org/downloads/standards/portable-stimulus