



# Python empowered GLS Bringup Vehicle

Debarati Banerjee

*Google India Private Limited, Bangalore, India*

[bdebarati@google.com](mailto:bdebarati@google.com)

Nikhil Singla

*Google India Private Limited, Bangalore, India*

[nikhilsingla@google.com](mailto:nikhilsingla@google.com)

Shantha B

*Google India Private Limited, Bangalore, India*

[shanthab@google.com](mailto:shanthab@google.com)

Pandithurai Sangaiyah

*Google India Private Limited, Bangalore, India*

[pandithurai@google.com](mailto:pandithurai@google.com)

**Abstract** - Gate Level Simulations(GLS) have always been one of the most critical activities in pre-Silicon verification. GLS helps in catching critical issues like incorrect MCP, incorrect power port connection, CDC issues etc which if missed could lead to Dead-on-Arrival chips. In majority cases, it's very difficult to catch these issues through normal RTL simulations. With the increasing complexity and size of designs, the challenges faced in GLS have also increased. A significant portion of the time is spent in getting the basic setup clean and flow flushing the netlist at all the GLS stages from Zero delay GLS to timing GLS. In this paper, we present a Python empowered GLS Bringup Vehicle(PEGV) to automate this process of generating GLS setup and verifying the critical paths in the design thus reducing the overall GLS turnaround time significantly. The solution generates an infrastructure which can be used to bring up the config and data path of any netlist with no dependency on functional testbench readiness and design expertise

## I. INTRODUCTION

One of the main challenges with GLS is turnaround time. With every netlist release, the main areas which takes time are :

- Sanitizing the collaterals
  - NRF, Syncflop and memory lists are not tool compatible
  - Non existing paths
  - Incomplete lib and standard cells list
- Stabilizing the DV setup
  - Wrong connections
  - Hardcoded RTL paths
  - Simulator switches and defines
  - Timing wrapper stabilization

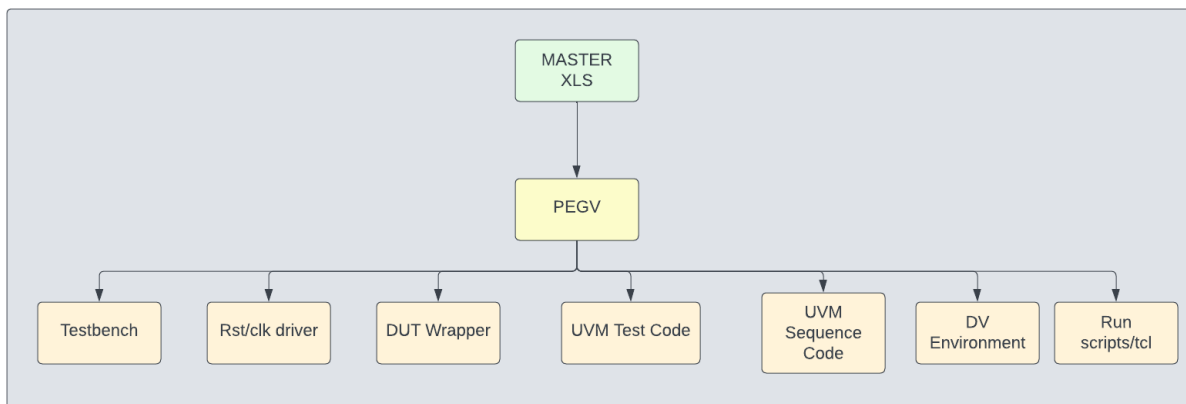
There is no standard approach to cater to above problems and bring up time is significantly high to get these initial issues resolved. We are proposing a standard solution which will fit into these gaps. PEGV is an automated solution

which takes in any netlist irrespective of the functionality and brings up the GLS test bench with bare minimum information required from the user in user-friendly ,easy to port XLS. Segregating the functional testbench from the bringup vehicle has enormous advantages. The key advantages of using the proposed solution are :

- Protocol agnostic: Engineer working on this solution need not be a protocol/DUT expert
- Simulators compatibility : Scripts are provided to make all the collaterals compatible to all industry standard simulators like Xcelium and VCS
- Faster closure : As the process is standard, GLS closure is very quick, where we don't see issues related to switches, defines and syntaxes
- Complete GLS solution: It covers all the vertices from zero delay to SDF sims
- Reusability : This approach can be used at all levels be it IP, subsystem or SOC level
- Fully automated solution: Once a user gives the required information to generate the infrastructure , the whole process is a single push button model
- No UVM/SV expertise required to work on this setup
- Master XLS : Single place for all the info. Any new release requires change at one place only
- Less manual error: As the whole process is automated, user dependent variables are reduced significantly, only input collaterals are required, rest all is standardized which reduces tool and flow related debugs thus allowing more time for focussing on the genuine design issues
- Basic Sanity Check by the PD team : Due to its portable and compact nature the PD team can use this package to run sanity checks on the netlist before releasing it to the DV team
- Trace : Once trace provided in the required format, datapath tests can also be run on this setup.
- Interface : The current setup supports standard interfaces like AXI,APB etc. The current scheme can be easily extended to any custom interfaces.

## II. METHODOLOGY

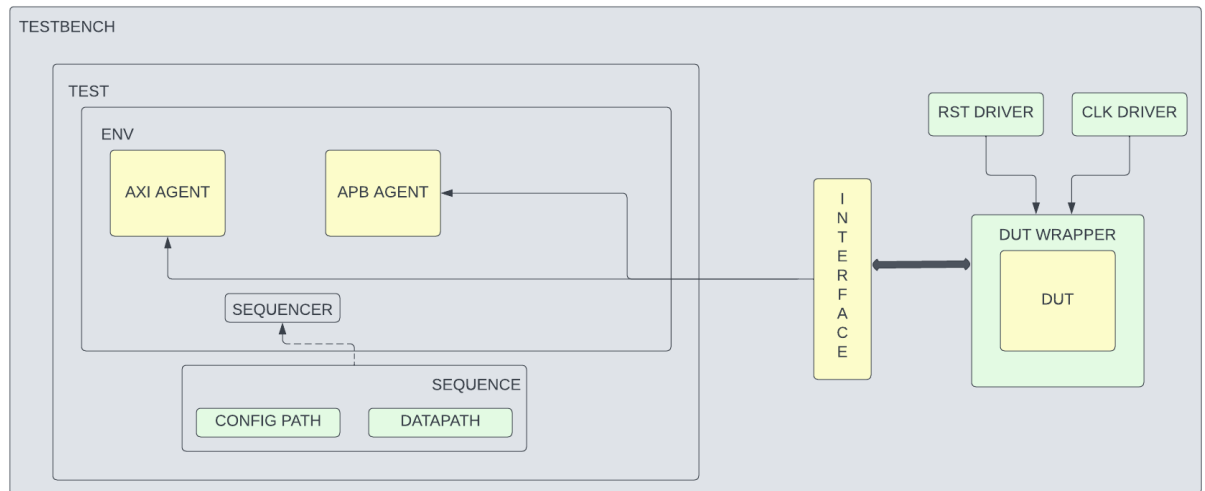
Figure 1 depicts the overall flow of PEGV. The master xls has all the required information. It is fed to the python empowered TB files generator to generate all the required files. Figure 1 depicts the overall PEGV flow



**Figure 1 : PEGV Flow**

The PEGV based GLS setup includes the following :

1. Testbench : The testbench has all the required DUT, VIP instantiation and relevant connections. The block diagram of the testbench used for the proposed solution is shown in figure 2.



**Figure 2 : Testbench**

Code Snippet of the testbench is shown in figure 3.

```

1 module dut_tb;
2
3   timeunit 1ns;
4   timeprecision 1ps;
5
6   import uvm_pkg::*;
7
8
9   initial begin
10    run_test("dut_base_test");
11  end
12
13  `include "tb_defines.svh"
14  `include "tb_rst_clk_drv.svh"
15  `include "inst_conn.svh"
16
17 endmodule

```

**Figure 3: Testbench code snippet**

2. Reset/Clock driver : Based on the information provided in the master xls, PEGV tool generates the required clocks and the resets. It also generates the file with the required connections between rst/clock and the dut.

```

12 // Rst/Clk Generation
13 initial begin
14     dut_rst_n = 0;
15     #100ns;
16     dut_rst_n = ~dut_rst_n;
17 end
18
19 initial begin
20     axi_vip_if_aresetn = `AXI_VIP_IF_ARESET_RST_VAL;
21     #100ns;
22     axi_vip_if_aresetn = ~axi_vip_if_aresetn;
23 end
24
25 initial begin
26     dut_ref_clk = 0;
27     forever begin
28         #1ns dut_ref_clk = ~dut_ref_clk;
29     end
30 end
31
32 initial begin
33     axi_vip_if_acl = 0;
34     forever begin
35         # `AXI_VIP_IF_AXI_CLK_TP axi_vip_if_acl = ~axi_vip_if_acl;
36     end
37 end
38

```

**Figure 4 : Rst/Clk Driver code snippet**

3. DUT Wrapper : This wrapper is generated based on the timing wrapper information provided by the user for timing gls. In timing GLS, the data or clk need to be delayed for proper latching of data at the destination node. The dut wrapper is used to provide this delay between the data and clk ports at the dut boundary. For non-timing GLS, it's a straightforward connection i.e. ports of dut wrapper are directly connected to dut ports.

```

1 always @(*)
2 begin
3     port0_dut <= #0.60ns port0_dut_wrapper;
4     port1_dut <= #0.60ns port1_dut_wrapper;
5     port2_dut <= #0.60ns port2_dut_wrapper;
6     port3_dut <= #0.60ns port3_dut_wrapper;
7     port4_dut <= #0.60ns port4_dut_wrapper;
8     port5_dut <= #0.60ns port5_dut_wrapper;
9     port6_dut <= #0.60ns port6_dut_wrapper;
10    port7_dut <= #0.60ns port7_dut_wrapper;
11 end

```

**Figure 5 : DUT Wrapper(timing gls) code snippet**

4. UVM Test Code : The UVM test code needed to verify the config path and datapath is part of this setup

```

01 //-----
02 task dut_base_test::run_phase(uvm_phase phase);
03     uvm_sequence seq_h;
04     `uvm_info(get_type_name(), "TEST : run_phase starts", UVM_LOW)
05
06     phase.raise_objection(this, "dut_base_test run_phase");
07     `uvm_info(get_type_name(), "TEST : raising objection", UVM_LOW)
08
09     seq_h = dut_base_vseq::type_id::create("seq_h", this);
10
11     `uvm_info(get_type_name(), "TEST : starting seq", UVM_LOW)
12     seq_h.start(m_env_h.m_virtual_sequencer_h);
13
14     `uvm_info(get_type_name(), "TEST : dropping objection", UVM_LOW)
15     phase.drop_objection(this, "dut_base_test run_phase");
16     `uvm_info(get_type_name(), "TEST : run_phase ends", UVM_LOW)
17
18 endtask: run_phase

```

**Figure 6 : UVM Test Code snippet**

5. UVM Sequence : PEGV setup supports testing of both config path and datapath. The base sequence has provision of verifying both the config path and datapath. The code corresponding to the config and data path is generated based on the information provided in the master xls. Code snippet of body() of the base sequence is shown in Figure 7.

```

44 //-----
45 task dut_base_vseq::body();
46     string test_nm;
47     `uvm_info(get_type_name(), "SEQ : start of body()", UVM_LOW)
48     #1ns;
49     if(!($value$plusargs("TEST_NAME", test_nm))) begin
50         test_nm = "CONFIG_ACCESS";
51     end
52     if(test_nm == "CONFIG_ACCESS") begin
53         access_dut();
54     end
55     else begin
56         acc_datapath(test_nm);
57     end
58     `uvm_info(get_type_name(), "SEQ : end of body()", UVM_LOW)
59
60 endtask: body
61 //-----
    
```

**Figure 7 : UVM Sequence code snippet**

6. DV Environment : This contains all the required agents and other relevant entities integrated. The VIP support can be easily extended to even custom based VIPs.

```

1 //-----
2 function void dut_env::build_phase(uvm_phase phase);
3     string inst_string;
4     super.build_phase(phase);
5
6     // Virtual Sequencer
7     m_virtual_sequencer_h = dut_virtual_sequencer::type_id::create("m_virtual_sequencer_h",
8                             this);
9
10    // AXI Master Agent
11    m_cfg_axi_manager_agent_h =
12        dut_axi_manager_agent_t::type_id::create("m_cfg_axi_manager_agent_h", this);
13 endfunction: build_phase
14 //-----
15 function void dut_env::connect_phase(uvm_phase phase);
16     super.connect_phase(phase);
17     set_vseqr(m_virtual_sequencer_h);
18 endfunction: connect_phase
19 //-----
20
    
```

**Figure 8 : DV Environment code snippet**

7. Run scripts/tcl : TCL files are generated with all the NRFs and memory deposits. The advantage of using tcl as compared to depositing it in sequence is reduction in overall execution time as any change in tcl doesn't need elaboration and compilation step. Only the test needs to be rerun. Apart from NRFs and memory deposits, tcl files to turn off timing checks on the sync flops are also generated. All these files are generated by the PEGV tool based on the input collateral files provided in the master xls by the user. These generated tcl files are compatible to all industry standard simulators like Xcelium and VCS

This generic solution is provided as a bundle which has following deliverables:

- User inputs : The User need to populate an XLS as per the design. This master xls has all the required information of the design, netlist, collaterals etc.

The xls has the following information -

- DUT ports- All the relevant ports of the DUT need to be populated in this sheet. This sheet also has the required information of clock,reset, any loopback/handshake signal ports. The polarity of signals also need to be populated in this sheet

	A	B	C	D
1	Dut Port Name	Type	Connection	
2	dut_clk1	clk,1ns		
3	dut_clk2	clk,2ns		
4	dut_rst	rst,0		
5	awaddr	axi_vip_if		
6	awlen	axi_vip_if		
7	awsize	axi_vip_if		
8	awburst	axi_vip_if		
9	awlock	axi_vip_if		
10	awcache	axi_vip_if		
11	scan_enable		1'b0	
12	o_req			
13	i_ack		`DUT.o_req	

**Figure 9 : DUT ports**

- Netlist and related collaterals - The path to the netlist and related collaterals like NRFs, memory instances, sync FF, io delay etc need to be populated appropriately in this sheet. The files mentioned in this sheet are used to generate all the required TB collaterals for GLS eg : deposit files, Tcheck files, timing wrapper etc.

	A	B	C	D
1	Collaterals	File Path		
2	LIB CELLS	lib_cells.f		
3	NRF	nrf.f		
4	MEM	mem.f		
5	SYNC CELLS	sync_cells.f		
6	NETLIST	netlist.v		
7	TIMING WRAPPER	timing_wrapper.f		
8				
9				
10				
11				
12				
13				

**Figure 10 : Collaterals**

- TB defines : Defines for hierarchies of different components and memories need to be populated in this sheet.

	A	B	C	D
1	Name	Value		
2	DUT	dut_wrapper		
3	AXI_TB_IF_INST	axi_if_inst		
4	AXI_VIF_ADDRESS_WIDTH	16		
5	MEM1	`DUT.inst1.ram0		
6				
7				
8				
9				
10				
11				
12				
13				

**Figure 11 : TB Defines**

- Stimulus information: In order to generate config path and datapath tests the following information need to be populated -
  - Config register path - This sheet contains all the required addresses that needs to be accessed in order to verify the config path

	A	B	C	D
1	Address	Mask	Prot	Type
2	34801000	hfff		AXI,APB
3	34901100	1		AXI
4	35A00000	bfff_ffff		
5	34901100	1		APB
6				
7				
8				
9				
10				
11				
12				
13				

**Figure 12 : Config test details**

- Datapath - Traces. The setup is built to replay the test cases generated in functional testbenches directly on this platform. All the accesses to DUT are captured in a format

called Trace which can be replayed by the GLS vehicle directly. Trace has all the hooks to support memory read/writes and including backdoor load/store. The traces required by this flow are portable and can be reused by SOC and subsystem teams as well.

	A	B	C	D
1	Testname	Trace Path		
2	TEST1	trace_1.c		
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				

**Figure 13 : Datapath test details**

- Python tool : The python tool takes the master xls as input and generates the standalone GLS TB setup. The files/entities generated using the tool are :
  - Clock & Reset Driver : It contains the clocks and reset driving logic.
  - UVM sequence Code : The sequence code to verify config path and data is generated by the tool.
    - Config path : Based on the address mentioned and the protocol information(eg : AXI,APB etc) mentioned in the xls, sequence code for reading and writing to the mentioned addresses is generated by the PEGV tool.
    - Data path : Replay driver to replay the traces generated by functional testbench. The datapath is verified using the traces. Once the trace is provided in the required format, datapath tests can also be executed using this PEGV. The following are the supported APIs-
      - LOAD\_MEM() : To load the contents of memory from a file
      - READ\_MEM() : To read the data from the required address of memory
      - WRITE\_MEM() : To write data at the required address of memory
      - POLL\_MEM\_DELAY() : To poll the required address of memory at a regular interval of time until the read data matches the expected data
      - POLL\_MEM\_NCNT() : To poll the required address of memory at a regular interval of time until the read data matches the expected data or the number of poll exceeds the specified limit whichever is earlier
      - DUMP\_MEM() : To write the contents of a memory onto a file
      - READ\_REG() : To read the data from the required register





- WRITE\_REG() : To write data at the required register
  - POLL\_REG\_DELAY() : To poll the required address at a regular interval of time until the read data matches the expected data
  - POLL\_REG\_NCNT() : To poll the required address at a regular interval of time until the read data matches the expected data or the number of poll exceeds the specified limit whichever is earlier.
- DUT connections : The connections of the DUT ports are also auto generated by the PEGV tool. The connections to TB reset, TB clks, TB interfaces, tie-offs, handshake connections, etc are all taken care of by the tool once the correct information has been provided by the user in the master xls.

### **III. APPLICATION**

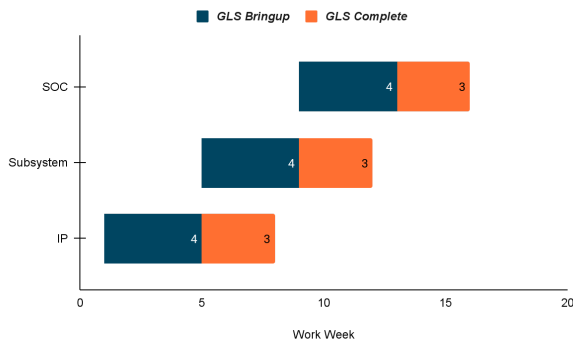
The generic nature of this solution makes it applicable to simple and complex systems. PEGV can be used not only by the DV team but also by the PD team to run sanity checks on the netlist before releasing it to the DV team thus reducing the number of iterations between DV and PD team. Any DV engineer can use this solution even without deep UVM knowledge, protocol expertise or design knowledge as the full flow of generating the DV setup is empowered by python automation. The automation aspect of this solution not only reduces the manual overhead significantly thus allowing more time and bandwidth for focussing on the genuine design issues but also enables parallel and early execution across IP, subsystem and SOC level. As the setup flow is similar across IP, subsystem and SOC level, subsystem teams can start with GLS bring up even before the IP team completes the GLS verification. Similarly the SOC team can start in parallel with the subsystem team on GLS. PEGV solution caters to the verification flow where users can start using this bring up vehicle with Zero delay sims which require very less information and as the design cycle matures, it can be extended for PG and timing sims.

### **IV. RELATED WORK**

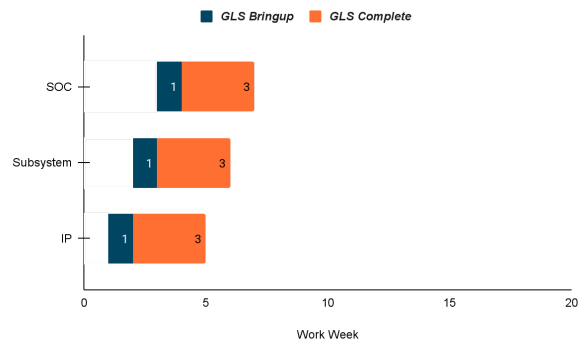
Conventionally, the same testbench is used for RTL verification and GLS.

### **V. RESULTS**

The idea is proven on multiple netlists and it has reduced the basic bringup time from 4 weeks to 3-4 days. As the setup overhead is similar across IP, subsystem and SOC level all these teams can start the GLS bringup in parallel. Figure 14 and figure 15 shows a comparative study of the time taken for GLS activity at 3 levels IP, Subsystem and SOC DV. Since the bringup time at each level is reduced in PEGV setup, the GLS turnaround time at all the levels have reduced tremendously. The reduction in turnaround time has a cascaded behaviour. For example, a subsystem depends on multiple IPs. If initial GLS bringup time of all the IPs are pulled low, then the point when GLS can start at subsystem level is reduced by a large factor. Similarly SOC GLS can be started at much earlier point compared to the conventional approach since bringup of all the subsystems is completed early using the PEGV setup.

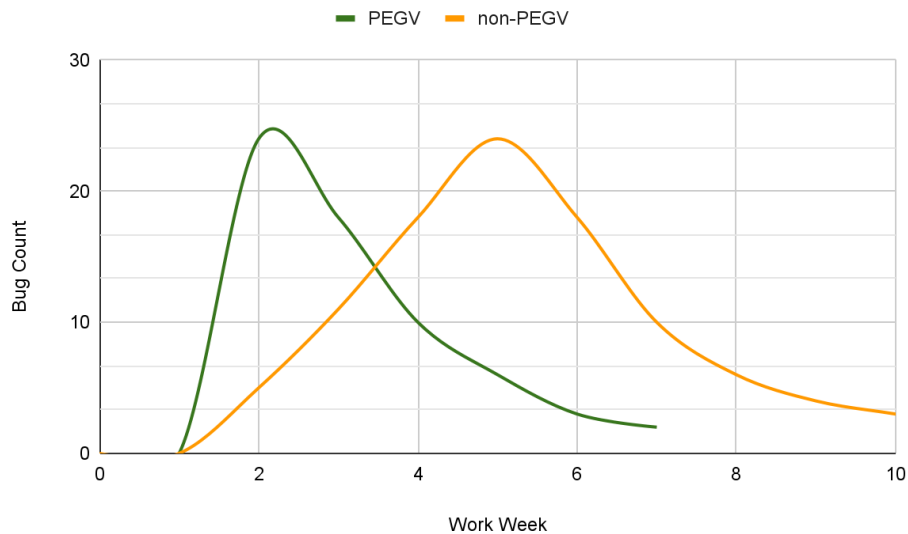


**Figure 14. Conventional Approach**



**Figure 15. PEGV Approach**

With PEGV, teams have found multiple issues related to wrong/missing NRFs, coldboot not working due to power switch connections very early in DV cycle without even consuming the netlist in functional block level testbenches thus reducing the GLS turnaround time significantly and enabling timely releases with higher confidence. Figure 16 shows a comparison between the bug chart for PEGV and non-PEGV approaches. The curve for PEGV is steeper as all the collaterals, TB, infra and simulator/tool related issues are identified much earlier than the conventional approach. Since GLS bringup time is reduced using PEGV, even genuine design issues can be uncovered in this period.



**Figure 16 : Bug Chart comparison between PEGV and Conventional Approach**

Even without being well versed with UVM or having in-depth design knowledge, this solution can be used by any DV engineer as once the information is correctly populated in the xls, the rest of the flow is automated. This solution can be used by the DV engineers even if it is not well versed with UVM and overall design knowledge. With every new release the DV setup overhead is minimal as the only change required is in the master xls file.



## **VI. SCOPE**

The vision is to make this setup a release checklist for all the netlist collaterals. All the netlists delivered will go through the sanity check on this platform before being plugged into the functional RTL testbench. Once made open source, any DV team could leverage this solution irrespective of the design complexity. PEGV supports both config and datapath verification. Currently it covers ~80% of the critical verification items. If all the traces can be provided in the required format, then even the complicated tests can also be executed on this setup thus leading to 100% verification closure on this setup.

## **VII. CONCLUSION**

The Python empowered GLS bringup vehicle proposed in this paper is a protocol agnostic and a generic solution that automates the GLS TB infra thus reducing the manual overhead of the DV team significantly. High flexibility and scalability of this solution enables it to be deployed for all kinds of design. Smaller GLS bringup time enables DV folks to spend more time on genuine design issues thus giving higher confidence before tapeout.