2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

Smart TSV (Through Silicon Via) Repair Automation in 3DIC Designs

Subramanian R, Senior Staff Engineer, Samsung Semiconductor India Research, Bengaluru, India (ramanian.r@samsung.com)

Jyoti Verma, Associate Director, Samsung Semiconductor India Research, Bengaluru, India (jyoti.verma@samsung.com)

Naveen Srivastava, Senior Staff Engineer, Samsung Semiconductor India Research, Bengaluru, India (naveen.sr@samsung.com)

Sekhar Dangudubiyyam, Associate Director, Samsung Semiconductor India Research, Bengaluru, India (sekhar.d@samsung.com)

***Abstract*— In recent years we have witnessed Semiconductor Industry reaching saturation point of making chips on as lowest as 3nm technology node with commendable PPA. Now the new emerging design trend is to make chips vertically stacked one on top of another and compacting all complexities over multiple dies such as Logic to Logic (High computing AI over APs) (OR) Logic to Memory dies (High computing AI/AP over stack of memories – HBM/LPDDR). The underlying principle of 3DIC technology is TSVs (Through Silicon Via) which connects multiple dies and carries critical data/control, clock and power signals from bottom die to top die and vice-versa. The Yield of 3DIC chips are heavily reliant on proper working conditions of TSVs. Any faulty TSV during chip manufacturing process will bring down the overall yield and cause chip's malfunction. Smart TSV fault repair is the need of the hour for 3DIC technology; capable of detecting and correcting faulty TSVs, thus improving the yield of 3DIC Fabs. This paper revolves around the key concept of TSV repair automation at multiple levels of Design Verification: Fault line detection, repair packet formation, repair of faulty TSV.**

Keywords—3DIC Design, TSV (Through Silicon Via), Testbench Automation

I. INTRODUCTION

In the era of high speed data computing, the need for smaller chips with high performance is growing higher with every passing day. In parallel, the complexity of the design and the number of transistors in a chip is also increasing. A novel method is needed to cater to both the above requirements and 3DIC design comes to the rescue as it not only accommodates large designs but also reduces the chip area by stacking one die on top of another thereby enabling faster data exchange.

For such designs, TSVs are the carriers of data between the dies and any fault in them poses a bigger challenge in the functioning of the chip. It becomes imperative to make sure the faulty TSVs are detected and corrected. With the ever-shrinking Design Verification cycle, the onus is on the Verification engineer to catch any bugs much early in the design cycle and to achieve that it requires smart ways to verify the TSV architecture in the design.

This paper talks about the smart approach to the verification of TSVs by means of automating all possible scenarios in the form of APIs which any user can use in their testbench environment according to their need.

II. RELATED WORK

TSV Repair architecture is developed to repair faulty TSVs in post silicon DIEs. As shown in Figure 1, Top and bottom dies are connected through different types of TSVs such as Power, Signal and few of the TSVs from TOP die form GPIO connections at BOTTOM die for Inter-Chip, Low Speed, High Speed, DRAM.

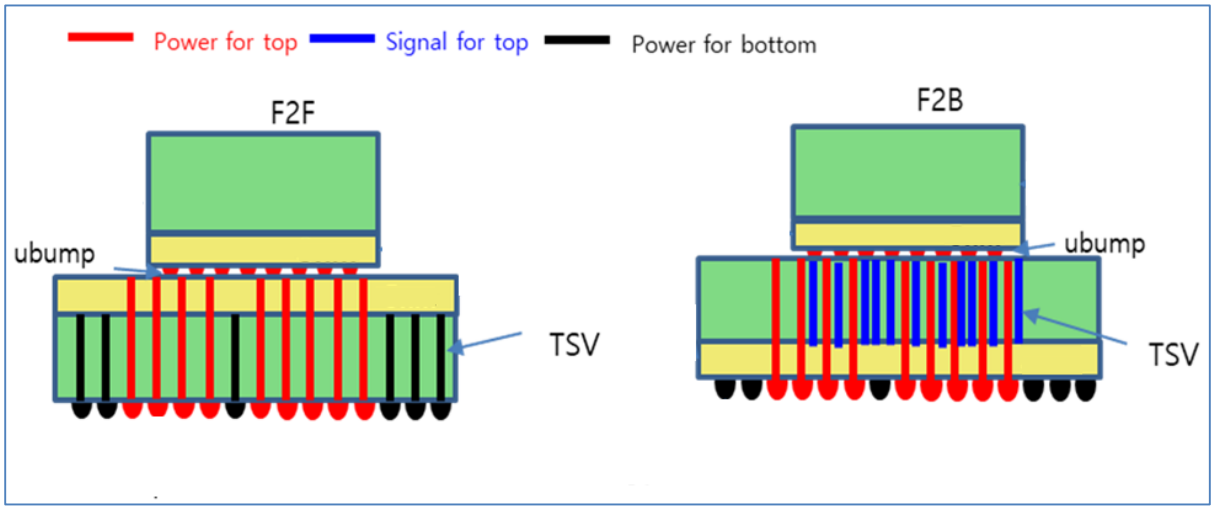

Figure 1: 3DIC Overview with TSV lines

A. *TSV Architecture in a typical 3DIC design*

A normal 3DIC SOC contains multiple dies with each die having multiple IPs and can have its own power manager. With the increase in number of IPs in each die, the number of TSV lines also increases. Also, in designs that have a split PMU architecture makes TSV lines more important as they carry vital control information and data between the dies as shown in Figure. 2.

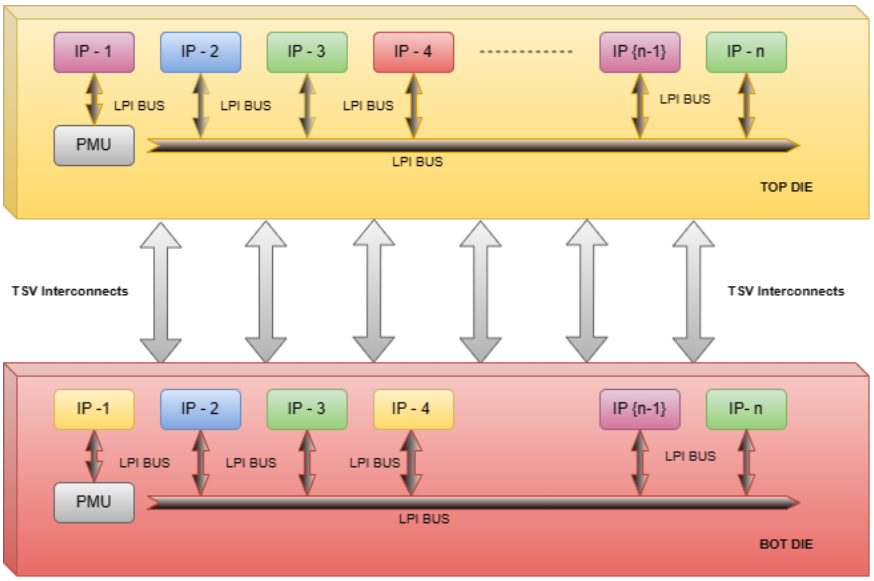

Figure 2: 3DIC SoC Architecture

B. Challenges in Conventional Verification of TSV Architecture in a SoC design

The conventional way for verification of the TSVs will be to have scenarios that have their datapath via these TSVs and talk between the dies. For e.g., if a Master CPU in the TOP die has to do a memory transfer to a RAM in the BOTTOM die, the user has to write a scenario that does Memory sweep of that. Also, this doesn't ensure the expected TSV lines are used. Additionally, the user has to write the code for injecting fault to check for faulty TSVs and ensure Repair mechanism happens. For a system with multiple IPs and multiple blocks that communicate using thousands of TSVs, this approach becomes laborious and time consuming involving multiple engineers. Also, with any increase/decrease in the number of TSV lines, the whole mechanism has to be repeated again for complete closure.

In summary, the below challenges are being confronted by the Verification Engineer in the Conventional method.

- Complete regression has to be run to see if all the TSV lines are hit. (Still coverage can be less than 100%)
- The engineer has to check the TSV lines his IP uses and add additional checks for fault detection and repair mechanism which can be tedious.
- The number of test scenarios to be written increases with any additional TSV line.
- Any bug will be caught late in the design cycle, that too if the specific datapath is being covered.
- Coverage metrics has to be generated separately.
- Holes in the Verification scenarios will be caught only when Toggle/Functional Coverage data is collected.

C. Smart Approach to the Verification TSV Architecture in a SoC design

To overcome the above said challenges, a smart verification environment is developed where the complete testbench is automated with the user having to fill only the necessary fields of the respective APIs and the rest of the verification is automatically taken care of. Along with that, it gives the user the flexibility to inject fault, repair and check the Repair signature in between the sequences and can be used exhaustively just by calling the APIs and no additional change to their existing code/sequence is required. This, in return, saves a lot of man hours and the process is simplified to a great extent.

The advantages seen with this smart approach is that

- A single testcase can cover the complete verification of TSV repair scheme in the design along with functional coverage metrics.
- This also helps the individual user to re-use the APIs in the sequences without having to re-compile the testbench thereby saving the simulation time
- Any change or update in the RTL Design requires update only to the Executable Spec (discussed in the section below) and no changes to the testbench code or environment is required.
- Targeted TSV line verification is also possible.
- Verification of TSV lines can be done IP-wise, BLK-wise or for the full chip based on the requirements.
- Easily scalable and portable between BLK-level and SOC-level Verification environment.
- Early bug catching in the design cycle.

III. IMPLEMENTATION

The Implementation of Smart TSV Repair Automation is described below. The primary requirement is the executable specification which contains the details of the TSV design information. Once that is fed to the TSV DB registry as shown in Figure 3, the testbench takes care of the rest in ensuring that all possible scenarios are checked for all the TSV lines in the design.

A. Summary of testbench Architecture

The overall Verification of TSV repair scheme can be decoupled into

a) Stack level

b) Cluster level and

c) SoC level

Figure 3 describes the automation flow deployed for Verification

Executable Spec: Holds key part of TSV design information such as number groups, TSV lines and design block in a CSV format.

TSV Repair Database: CSV file is processed into registry (set of arrays queried by keys) for fault injection, correction and pattern generation.

TSV Sequencer: Generates connectivity, Fault Injection, repair and repair packet generation sequences

Connectivity Check: Ensures the signal from the TOP die is connected to the right signal on the BOTTOM die (and vice-versa) as given in the Executable spec.

Fault Injection: Injects stuck-at-0 or stuck-at-1 faults at the Driver side of the TSV and the same has to be detected at the receiver side on the other die. Also the position of the faulty bit in the TSV group is also identified. With this, datapath through the faulty TSV will show mismatch.

Fault Repair: Fault is injected as described above and when "*repair enable*" is set, the fault is repaired automatically by shifting the data bit to the adjacent functional TSV line and the last bit uses the spare TSV line in the group. With this, datapath through the repaired TSVs will show a match.

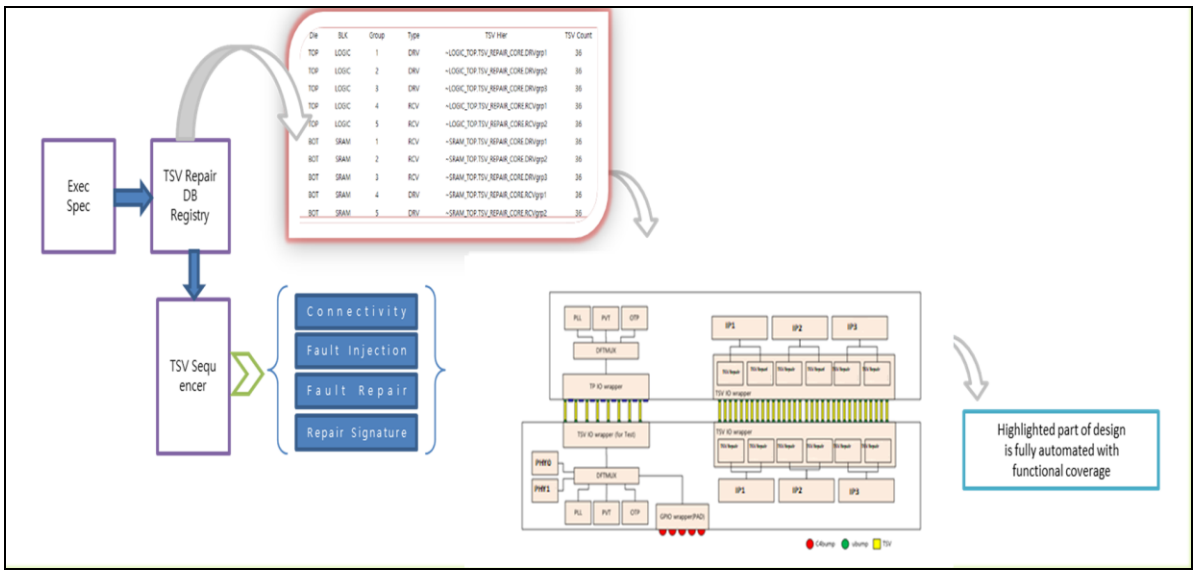

Figure 3: TSV Repair Automation Flow

The above TSV repair automation flow is extensively used in 3DIC flow to verify:

- Connectivity of nearly ~2000 TSV lines from Bottom to Top die and vice versa
- Random fault injection (Stuck-At-Zero, Stuck-At-One) and detection at Bottom/Top dies with accurate fault position detection
- Fault repair key generation and fault removal checks
- TSV Repair packet generation and repair signature propagation checks from OTP controller to destination blocks
- IP/BLK functional data path with random TSV fault injection and repair

B. Single Testcase Implementation

To make things simpler, a single testcase that covers all the above scenarios have been written such that it does connectivity check, fault injection and repair, repair signature propagation check for all the TSV lines at one go. This helps in reducing the number of testcases to be run and number of iterations to be done to get 100% coverage.

C. Connectivity Check Implementation

Though a single testcase covers the complete set of scenarios, it also provides flexibility to the user to use it in a specified manner with multiple combinations of Fault Injection, Repair Enable/Disable, Wrong Signature of TSV lines, etc. The Implementation of Connectivity check is described in the below piece of code as shown in Figure 4. The API is robust enough to take arguments in decreasing order of size. The key points of this API are:

- If the API is called without any arguments, the connectivity check is performed for all the TSV lines in all the IPs of all the BLKs in all the DIEs.
- If only the DIE information is specified, the API performs connectivity check for all the TSV lines in all the IPs of all the BLKs in that DIE.
- If the DIE and BLK information is specified, the API performs connectivity check for all the TSV lines in all the IPs of the specified BLK in the specified DIE.
- Connectivity check for specific TSV line in an IP of a BLK in a particular DIE also can be done.

```
virtual task tsv_repair_connectivity_check(string die = "UNDEFINED", string blk ="UNDEFINED",  int group = -1, int tsv_line = -1);
  //Variable declarations

  //--------------- Flexibility use this task Sequence control or thru plus args  ----------------
  user_blk = (top_tsv_dk_groups.exists(blk) | bot_tsv_dk_groups.exists(blk));
  if (user_blk) `uvm_info(get_type_name(), $psprintf("User Specified block -%0s to be run for connectivity check ",blk), UVM_LOW)
    `uvm_info(get_type_name(), $psprintf("User Specified block -%0s to be run for connectivity check ",blk), UVM_LOW)

  // DRV ---> RCV  - Simple connectivity check, drive random data on DRV and check RCV
  //GEt the DIE_NAME info.
  foreach (``DIE_NAME``_tsv_dk_groups[_a]) begin // For every group present in the DB registry
    if (user_blk && _a != blk) continue;  // If user specified block (either by task arg or testplus args) then do connectivity for that block only
      foreach (top_tsv_dk_groups[_a].drv_group[_b]) begin
          uvm_info(get_type_name(), $psprintf("BLK:%s - Connectivity Check for Group %0d", _a, _b), UVM_LOW)
          path = ``DIE_NAME`` tsv_dk_groups[_a].drv_group[_b];
          tsv_lines = tsv_count[path];
          // Get Cross-die mapping info ;
          foreach(loop[_x]) begin
            //----------Logic to drive on DRV line----------------
            //----------Logic to check on RCV line----------------
            if (write_data != read_data ) begin
              `uvm_error(get_type_name(), $psprintf("%s TSV_COUNT (%0d): (%0h) != (%0h) %s", path, tsv_lines, random_data, read_data, path1))
            end
            else begin
              `uvm_info(get_type_name(), $psprintf("TSV Connectivty check PASSED %s TSV_COUNT (%0d): (%0h) == (%0h) %s", path, tsv_lines, random_data, read_data, path1), UVM_LOW)
            end
          end ; // End of for loop[$]
      end
    end
  endtask
```

Figure 4: Sample Code snippet showing the Connectivity Check Implementation

D. Fault Injection and Repair Check Implementation

Similar to Connectivity check, the testbench has APIs to inject Fault, detect the fault and Repair the same based on control signals. As mentioned above, the user can call the APIs in between sequences to implement the same for the TSVs which they work with. The same has been shown in the code snippet in Figure 5. As can be seen, datapath is accessed before fault injection. Then using the common API- *tsv_InjectFault*, the user can inject fault on the specific tsv_line of a BLK inside the DIE. Then again, datapath is accessed which is expected to FAIL.

The subsequent part of the code shows the Repair Check Implementation. Using the API *tsv_SetRepairEnable* and *tsv_SetRepairSignature*, the repair configurations are done for the TSV line. Post which the Faulty TSV line is corrected and when the datapath access is called again, it is expected to PASS. Once this is done, we can disable the RepairEnable and release the Repair Signature by calling the APIs – *tsv_SetRepairEnable, tsv_ReleaseFault, tsv_RelRepairSignature.*

```
        `uvm_info(get_type_name(), $psprintf("[TSV SEQ] :: BEFORE FAULT INJECTION"), UVM_LOW)
        axi_access(addr1,data1);   //User sequence
        axi_access(addr2,data2);   //User sequence

tb.tsv_repair_infra_seq.tsv_InjectFault(``DIE``,``BLK``,tsv_group,tsv_line,stuck_at_0);

        `uvm_info(get_type_name(), $psprintf("[TSV SEQ] :: AFTER FAULT INJECTION"), UVM_LOW)
        axi_access(addr1,data1);  //User sequence
        axi_access(addr2,data2);  //User sequence

tb.tsv_repair_infra_seq.tsv_SetRepairEnable(``DIE``,``BLK``,tsv_group,enable);
tb.tsv_repair_infra_seq.tsv_SetRepairSignature(``DIE``,``BLK``,tsv_group,tsv_line);

        `uvm_info(get_type_name(), $psprintf("[TSV SEQ] :: AFTER FAULT REPAIR"), UVM_LOW)
        axi_access(addr1,data1);  //User sequence
        axi_access(addr2,data2);  //User sequence

#10us;
tb.tsv_repair_infra_seq.tsv_SetRepairEnable(``DIE``,``BLK``,tsv_group,disable);
tb.tsv_repair_infra_seq.tsv_ReleaseFault(``DIE``,``BLK``,tsv_group,tsv_line);
tb.tsv_repair_infra_seq.tsv_RelRepairSignature(``DIE``,``BLK``,tsv_group,tsv_line);
```

Figure 5: Sample Code snippet showing the usage of APIs inside user sequences

E. Functional Coverage Implementation

This Smart testbench environment comes with Functional Coverage Implementation to measure the extent of TSV Repair Check Verification. Some of the major cover points that have been used are listed below:

- Faulty lines – This coverpoint is used to make sure all the TSV lines are checked for Fault Injection for both stuck-at-0 and stuck-at-1
- Signature based Repair check – This coverpoint covers the TSV line repair based on the control signals and the Repair Signature for all the BLKs which are pre-loaded in OTP. This gives the complete coverage with the combination of TSV line and its control signal for Repair enable and the repairing of the same based on Signature.
- Fault pattern – Additional coverpoint is to check the fault pattern being injected with 0xAA, 0x55 and 0xFF patterns being the bins.

IV. RESULT

In a conventional approach involving all test-suites that cover the entire TSV lines, requires a full regression to be run. In this automated approach, the same can be verified with full functional coverage in less than a day.

Approach	Test Suits required	DV Effort
Conventional	Full regression suite that covers all TSV lines between the dies	3-4 weeks
Fully Automated	Single testcase covering all TSV lines	< 1 day

Also, we were able to achieve 100% Functional Coverage and 100% Toggle Coverage with the single testcase that covers the entire TSV lines (~2000 lines) as described in Section II Sub-Section B.

This helped in catching critical connectivity miss in the design where the fault injected at the DRV side was not caught at the RCV side, as it primarily used the Executable Spec as the input.

V. CONCLUSION

With advanced technology nodes, reduced power consumption, complex designs and a compressed verification cycle, Smart TSV Repair Automation framework provides a one-stop verification approach to cover all possible scenarios within a smaller timeframe, provides additional check in the form of in-built Functional Coverage, flexibility to be re-used by any user without having to change the sequences much and catch bugs early in the cycle thereby enabling a smoother and error-free communication between the dies in a 3DIC system.