Tackling Register Aliasing Verification Challenges in Complex ASIC Design

Shan Yan, Jie Wu, Jing Li
Broadcom Limited
shan.yan@broadcom.com, jie.wu@broadcom.com, jing-r.li@broadcom.com

***Abstract*- Register design and implementation is one of the most important parts of today's complex ASIC designs. Register verification is a significant part of the design verification problem. Register aliasing issues are very hard to capture, with hundreds or thousands of registers implemented in a single chip. With the traditional simulation-based verification method for finding the register aliasing issue, it is usually hard to achieve coverage completeness and is very time consuming. In this paper, we propose a register verification methodology that combines assertion-based technique and simulation to find register aliasing issues. Our methodology comprises two parts. First, for each register module in the design, an assertion model is developed to check read_enable and write_enable signals for the register access. An overall read_enable counter and write_enable counter are maintained at the chip level to count register access on all registers. The assertion models for all register modules are bound into all the instances of that module. The assertion models are to check that no more than one read or write access happens to any register instance at any given time. Second, three register access sequences are carefully designed to test the register read and write accesses to make sure all register aliasing issues can be captured. All sequences use the UVM Register Access Layer (RAL) model and combine both frontdoor and backdoor accesses. To make our methodology applicable to complex designs, the whole procedure is automated. We applied this methodology to three projects within our high-speed interconnect physical layer products. Several register aliasing bugs were found in the earlier stage of the design. The performance has also significantly improved to O(n) by using our methodology, compared to the traditional simulation-based register aliasing verification method of $O(n^2)$.**

I. INTRODUCTION

Register design and implementation is one of the most important parts of today's hardware and IC designs. Registers contain the configuration setting of the hardware and are the basis of the hardware and software interface. Register verification is a significant part of the design verification problem. It is one of the first aspects of the design that must be tested because the rest of the semiconductor functionality depends on the accuracy of the register implementation.

In hardware design, register aliasing describes two possible situations. One is that a register location in the hardware design can be accessed through different symbolic names and addresses, as shown in Figure 1(a). Thus, modifying the value in that location through one name implicitly modifies the values associated with all aliased names. The other situation is that one symbolic name and address is associated with multiple register locations in the hardware design, as shown in Figure 1(b). Thus, modifying the value through one name and address modifies the values of registers associated with that name. Register aliasing can either be a hardware design choice or a hardware failure. In a failure, register aliasing problems are usually caused by hardware design bugs, such as address decoding logic errors, one or more address bits being shorted together, or being forced to ground (logic 0) or to the supply voltage (logic 1). In these cases, the register accesses through symbolic names or addresses cannot reach the locations intended or may reach the wrong location [1].

Register aliasing issues are very hard to capture for today's complex ASIC designs with hundreds or thousands of registers implemented. The traditional simulation-based verification method for finding the register aliasing issue is usually based on the methodology of writing a unique value to one register and reading back the values from all registers and then comparing. However, this methodology is very time consuming because all the registers inside the

whole SoC have to be tested at once and can't be split by blocks and tested in parallel. But otherwise, the issue cannot be captured. This makes the simulation time $O(n^2)$, where n is the total number of registers in the whole SoC design. Also, the sequence and order of how the registers are written and read have to be carefully designed or the issue may not be captured correctly and the coverage completeness cannot be reached.

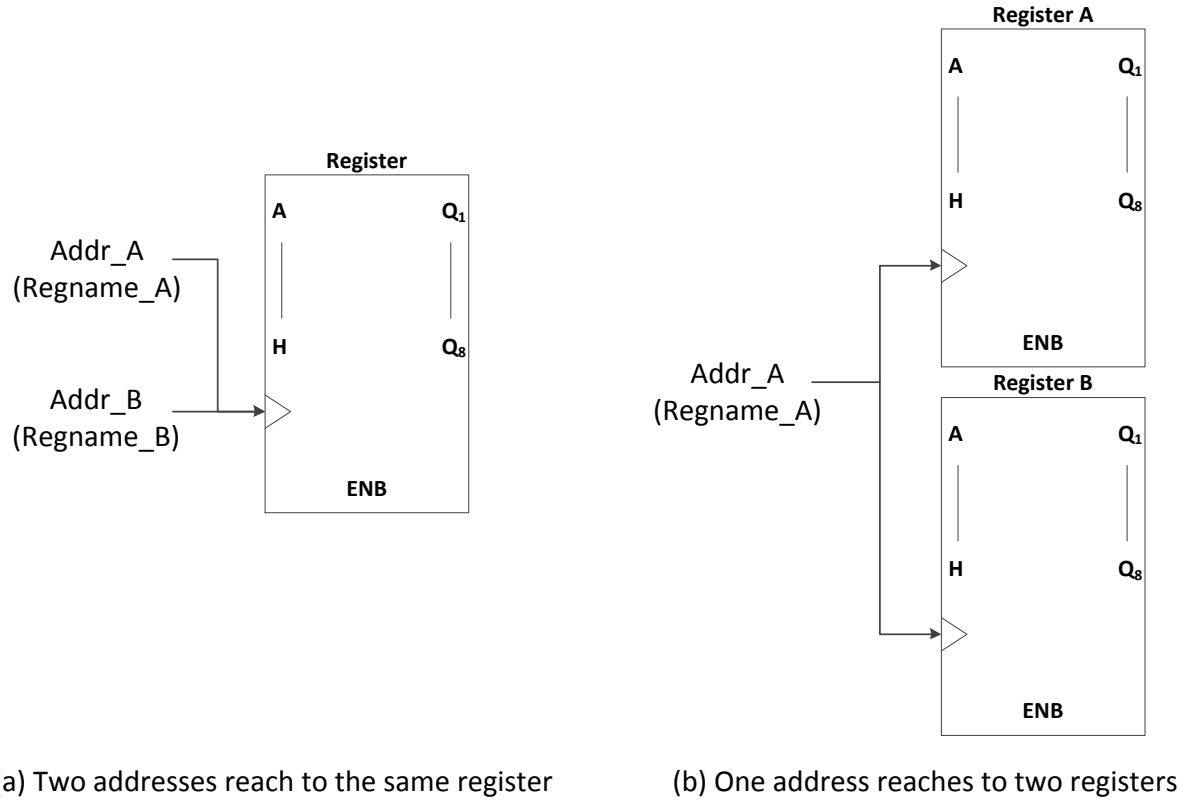

Figure 1. Register aliasing scenarios

In this paper, we propose a register verification methodology that can achieve register verification coverage completeness and easily detect register aliasing issues. Our methodology combines assertion-based technique and simulation, and thus can achieve great performance improvement over the traditional pure simulation-based verification methodology. Our methodology comprises two parts. First, for each register module in the design, an assertion model is developed to check read_enable and write_enable signals for the register access. An overall read_enable counter and write_enable counter are maintained at the chip level to count register access on all registers at given time. The assertion models for all register modules are bound into all the instances of that module. When register read or write access happens to a specific instance, the corresponding read_enable and write_enable are asserted, and the read_enable or write_enable counter is counted. If the counters are counted to more than 1 at any time during simulation, it means there are accesses to more than one register at the same time. Thus, the register aliasing issue is captured. With the assertion information reported, the aliased registers are easily found and reported.

After this, three register sequences are carefully designed to test the register read and write accesses. All sequences use the UVM Register Access Layer (RAL) model and combine both frontdoor and backdoor accesses. The first two sequences are based on UVM RAL backdoor access. One sequence is a backdoor write to a register and then a frontdoor read from that register, and so on for all the registers. The other sequence is a frontdoor write to one register and then a backdoor read from that register, and so on for all the registers. The hierarchical paths to the registers per design specifications are provided for the backdoor access. Using these two sequences, the correctness of the decoding logic and access paths can be verified for all registers. The first two sequences make sure each read and write can reach to the correct location in the hardware. With this goal achieved, the third sequence of frontdoor writes and frontdoor reads of each register is used to test register aliasing. This step ensures each read and write does not reach the unwanted locations in the hardware. Each read and write only need be done once, thus reducing overall simulation time to O(n). Furthermore, with the assertions in place, the tests can be broken down to run on a per-block basis, and all the blocks can be tested in parallel to further save simulation time.

To make our methodology applicable to complex designs, we automated the whole procedure. In our designs, an internal format-RDB is used to define the registers. These RDB files are used to generate both the UVM RAL model and the register blocks in the hardware. Thus scripts are developed to use the same register naming rule to automatically extract the registers' RTL hierarchical paths and put them into the UVM RAL model for backdoor access. Tests are also automatically generated for each design block to run the three sequences.

We applied this methodology to three projects of our high-speed interconnect physical layer products. Each of our chips contains thousands of registers. Several register aliasing bugs were found in the earlier stage of the design, giving us the confidence for a bug-free design and tapeout. Also, the performance has significantly improved by using our methodology. Using traditional register testing sequences, each simulation would run for a couple of days depending on the types of the register interfaces. Now, with each test running on a single register block and multiple tests running in parallel on the LSF farm, all tests can finish within hours.

The rest of the paper is organized as follows. Section II outlines the related work. Section III presents our assertion-based register aliasing modelling and methodologies in detail. Section IV presents the three register access sequences that are designed to test the register accesses. Results of applying our methodologies to our projects are shown in Section V, and the conclusion is presented in Section VI.

II. PREVIOUS WORK

The traditional verification method for finding the register aliasing issue is usually based on the methodology of writing a unique value to one register, reading back the values from all registers, and then comparing to detect register aliasing[2]. However, this methodology is very time consuming because all the registers inside the whole SoC have to be tested together. They cannot be split by blocks and tested in parallel. But otherwise, the issue cannot be captured. This makes the simulation time for running such sequence be $O(n^2)$, where n is the total number of registers in the whole SoC design. Since today's design complexity has become greater and greater, ASIC chips usually contain hundreds or thousands of registers. Such a method is hardly applicable to those designs. Further, the sequence and order of how the registers are written with random value and read back have to be carefully designed. Otherwise, the issue may not be captured correctly and the coverage completeness cannot be reached.

III. ASSERTION-BASED REGISTER ACCESS MODELLING

To efficiently detect register aliasing issues and achieve register verification coverage completeness in complex ASIC designs, we propose a register verification methodology that combines assertion-based technique and simulation. Our methodology comprises two parts. First, an assertion-based register access model is developed for each type of register module in the design. The details are presented in this section. Second, three simulation sequences are carefully designed to test the register read and write accesses. Those details are presented in next section. In combination, our methodology can achieve great performance improvement over the traditional pure simulation-based verification.

In ASIC designs, several standard or home-developed register modules would be used and instantiated to implement registers of various types. For each module type, it would commonly have clock, reset, default value, register address, write_data and read_data inputs, and read_data outputs, as shown in Figure 2. It also has read_enable and write_enable signals to control the register read and write accesses. Our assertion model is developed for each register module type to check the read_enable and write_enable signals for register access. For our designs, the bus architecture is used, and all the registers are attached to the same bus for access. Thus, at any given time, only one register is accessed by the master on the bus. An overall read_enable counter and a write_enable counter are maintained at the chip level to count register access on all registers. The assertion models for all register modules are bound into all the instances of that module. When register read or write access happens to a specific instance, the corresponding read_enable and write_enable are asserted, and the read_enable or write_enable counter is counted. If the counters are counted to more than 1 at any time during simulation, it means there are accesses to more than one register at the same time. Thus, the register aliasing issue is captured. With the assertion information reported, the aliased registers are easily found and reported.

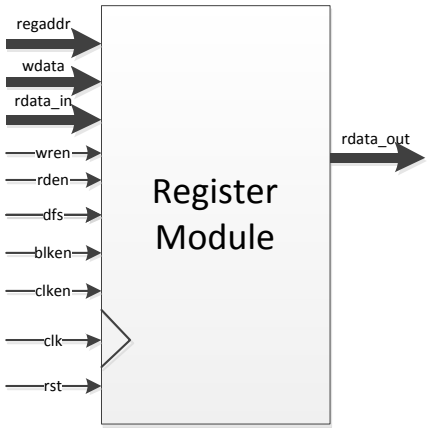

Figure 2. Register module

Figure 3 and 4 show the code of the assertion model.

Figure 3 shows how the read_enable(rd_en) and write_enable(wr_en) signals of typeA register are checked and counted when they are asserted. Lines 11-22 show how the rd_en and wr_en are captured. In this register type, the read operation can always finish in one cycle, and thus rd_en is asserted for only one clock cycle and then

deasserted. For the write operation, it may take more than one cycle before the write is done. To make sure the assertion of the wr_en signal is only counted once, an extra register reg_wr_d is used (lines 16-22). Two counters, reg_wcnt and reg_rcnt, are used to record the read count and write count of the registers at the chip level. They are increased whenever the rd_en or the wr_en is asserted (lines 24-39).

```
1   module typeA_reg_bind (clk, dfs, regad, data_in, wr_en, rd_en, resetb, data_out);
2
3   //------------------------------------------------------------------------------
4   // Port Declarations                          Signal Description
5   //------------------------------------------------------------------------------
6   input                    clk, resetb;
7   input                    wr_en, rd_en ;          // write enable, read enable
8   input     [15: 0]        dfs, data_in, data_out; // default register state, input data, output data
9   input     [ 3: 0]        regad ;                 // address
10
11  reg reg_wr_d;
12
13  assign reg_wr = wr_en & regen;
14  assign reg_rd = rd_en & regen;
15
16  always @(negedge clk)
17  begin
18    if (reg_wr) reg_wr_d = 1;
19  end
20
21  always @(negedge reg_wr)
22    reg_wr_d = 0;
23
24  always @(posedge clk)
25  begin
26    if (reg_wr & ~reg_wr_d & `TB_TOP.reg_alias_check) begin
27      $display ($time, "%m : reg_wr is asserted.");
28      `TB_TOP.reg_wcnt = `TB_TOP.reg_wcnt + 1;
29    end
30  end
31
32  always @(posedge reg_rd)
33  begin
34    #1ns; // filtering glitch
35    if (reg_rd & `TB_TOP.reg_alias_check) begin
36      $display ($time, "%m : reg_rd is asserted.");
37      `TB_TOP.reg_rcnt = `TB_TOP.reg_rcnt + 1;
38    end
39  end
40
41  endmodule
```

Figure 3. TypeA register module read_enable and write_enable check model example

```
1   bit [15:0] reg_wcnt, reg_rcnt;
2   bit reg_rd_chk_on, reg_wr_chk_on;
3   bit reg_alias_check;
4
5   initial
6   begin
7     reg_wcnt = 0;
8     reg_rcnt = 0;
9     reg_wr_chk_on = 1'b0;
10    reg_rd_chk_on = 1'b0;
11    reg_alias_check = 1'b0;
12    if ($test$plusargs("reg_alias_check")) begin
13      reg_alias_check = 1;
14    end
15  end
16
17  always @(posedge reg_alias_check) begin
18    reg_wcnt = 0;
19    reg_rcnt = 0;
20  end
21
22  always @(reg_wcnt)
23  begin
24    assert(reg_wcnt <= 1)
25    else `uvm_error(get_type_name(), $psprintf("!ERROR! : %m reg_wcnt %d > 1 for unicast register
    when reg_wr_chk_on is 1", reg_wcnt));
26  end
27
28  always @(reg_rcnt)
29  begin
30    assert(reg_rcnt <= 1)
31    else `uvm_error(get_type_name(), $psprintf("!ERROR! : %m reg_rcnt %d > 1 for unicast register
    when reg_rd_chk_on is 1", reg_rcnt));
32  end
33
34  always @(posedge reg_wr_chk_on) begin
35    reg_wcnt = 0;
36    reg_rcnt = 0;
37  end
38
39  always @(posedge reg_rd_chk_on) begin
40    reg_wcnt = 0;
41    reg_rcnt = 0;
42  end
43
44  bind typeA_reg_module typeA_reg_bind u_typeA_reg_bind_inst(.*);
45  bind typeB_reg_module typeB_reg_bind u_typeB_reg_bind_inst(.*);
46  bind typeC_reg_module typeC_reg_bind u_typeC_reg_bind_inst(.*);
```

Figure 4. Example code for counters, assertions and binding.

```
1   task write_register (intfType intf, string regname = "undef", bit[31:0] wdata, regIntfType reg_if =
    APB);
2     .....
3     reg_inst = get_reg_by_name (intf, regname);
4     bit ok = 1'b1;
5     ok &= uvm_hdl_deposit("system.reg_wr_chk_on", 1'b1);
6
7     reg_inst.set(wdata);
8     reg_inst.write(status, wdata, , regif2name(reg_inst, reg_if));
9
10    ok &= uvm_hdl_deposit("system.reg_wr_chk_on", 1'b0);
11  endtask


12  task read_register (intfType intf, string regname = "undef", output bit [31:0] rdata, input regIntfType
    reg_if = APB);
13    .....
14    bit ok = 1'b1;
15    ok &= uvm_hdl_deposit("system.reg_rd_chk_on", 1'b1);
16
17    reg_inst = get_reg_by_name (intf, regname);
18    reg_inst.read(status, rdata,,regif2name(reg_inst, reg_if));
19    reg_inst.set(rdata);
20
21    ok &= uvm_hdl_deposit("system.reg_rd_chk_on", 1'b0);
22  endtask
```

Figure 5. Example register read/write tasks using RAL

Figures 4 and 5 illustrate how the two counters are counted and checked for register aliasing errors. In Figure 4, the assertion models for all register modules are bound into all the instances of that module. In our example, three register modules are used, and the corresponding models are bound to them accordingly (lines 44-46). The overall reg_wcnt and reg_rcnt counters are checked whenever there are changes using assertions. When there are register aliasing errors, the assertions fail immediately, and the problematic registers are also reported so that they can easily be located (lines 22-42).

For our methodology to work correctly, the counters need to start counting at the beginning of any register access and need to stop counting and reset before the beginning of the next register access. The latter part is very important because it makes sure that no extra register access happens when it shouldn't. This is controlled by two bits, reg_wr_chk_on and reg_rd_chk_on. These two bits are set and reset inside the UVM Register Access Layer (RAL) model by modifying the register read() and write() tasks. In Figure 5, two tasks, write_register and read_register, are implemented. At the beginning of the tasks, the reg_wr_chk_on or the reg_rd_chk_on bit is set to 1, and at the end of the tasks, the reg_wr_chk_on or the reg_rd_chk_on bit is reset to 0. The reg_wcnt or reg_rcnt is reset to 0 only at posedge of reg_wr_chk_on and reg_rd_chk_on (lines 34-42) to make sure that invalid access can be captured not only during the expected register access but also when there should be no register access at all.

IV. REGISTER ACCESS SEQUENCES

To capture all the register aliasing issues in the design, three register read and write sequences are carefully designed to work with our register read_enable and write_enable check models. All sequences use the UVM Register Access Layer (RAL) model and combine both frontdoor and backdoor accesses. Backdoor accesses are extensively used to reduce simulation time and improve efficiency. The first two sequences are based on UVM RAL backdoor and frontdoor accesses. The third sequence is based on UVM RAL frontdoor access only. The hierarchical paths to the registers per design specifications are provided for the backdoor access. All sequences use walking-ones and walking-zeros subsequences to test every valid bit of each register[3].

A. Backdoor write and frontdoor read sequence

In this sequence, for each register in the design, every valid bit is tested using walking-ones and walking-zeros subsequences. The value is first written to the register through the backdoor and then read back via the frontdoor to check correctness. This sequence ensures each read is from the correct location in the design and does not reach to unwanted locations in hardware. This sequence checks only Readable and Writable (RW) registers and Read-Only (RO) registers. It can't check Write-Only (WO) registers. Both decoding logic and register aliasing logic related to read operation can be captured using this sequence.

B. Frontdoor write and backdoor read sequence

In this sequence, for each register in the design, every valid bit is tested using walking-ones and walking-zeros subsequences. The value is first written to the register through the frontdoor and then read back via the backdoor to check correctness. This sequence ensures that each write can reach to the correct location in the design and does not reach to unwanted locations in hardware. This sequence checks only RW registers and WO registers. It can't check RO registers. Both decoding logic and register aliasing logic related to the write operation can be captured using this sequence.

C. Frontdoor write and frontdoor read sequence

In this sequence, for each register in the design, every valid bit is tested using walking-ones and walking-zeros subsequences. The value is first written to the register through the frontdoor and then read back via the frontdoor as well to check correctness. This sequence checks only RW registers. It can't check RO and WO registers.

By using sequences A and B, the correctness of the decoding logic and access paths can be verified for all registers. This sequence ensures no potential bugs were introduced due to any potential behavior discrepancy between frontdoor accesses and backdoor accesses.

For all three sequences, each read and write only need be done once, thus reducing overall simulation time to O(n). Furthermore, with the assertions in place, the tests can be broken down to run on a per-block basis, and all the blocks can be tested in parallel to further save simulation time.

V. RESULTS

To make our methodology applicable to complex designs, we automated the whole procedure. In our designs, an internal format-RDB (Register Database Format) is used to define the registers. These RDB files are used to

generate both register blocks in the design and a UVM RAL model in verification. Several scripts are developed to use the same register naming rule to automatically extract the registers' RTL hierarchical paths in the design and put them into the UVM RAL model for backdoor access.

Tests are also automatically generated for each design block to run the three sequences.

We applied this methodology to three projects of our high-speed interconnect physical layer products. Each of our chips contains thousands of registers. Several register aliasing bugs were found in the earlier stage of the design, giving us the confidence for a bug-free design and tapeout.

A. Address decoding error

The first type of register aliasing bug we found was due to the address decoding logic design errors. The example scenario is shown in Figure 6. Register A with address AddrA was not overlapped with Register B with address AddrB in the design specification and RDB. However, due to the decoding logic design error, these two registers can both be accessed through AddrA, but AddrB cannot access Register B. This type of error can be found using a traditional register aliasing checking sequence discussed in the previous section. However, this requires a very long simulation time. This type of error could also possibly be captured using the three sequences alone without the read_enable and write_enable checking model. Though it's not guaranteed and depends on the real design scenario. For example, if RegisterA and RegisterB have the same default value, and AddrB is wrongly designed to access another register with the same default value, then using those sequences alone only once cannot capture the issue because the read value could happen to be the same as the write value. Several iterations of the sequences with randomizations may need to be run to capture the problem. But when using the read_enable and write_enable checking, this type of bug is guaranteed to be found in one run.

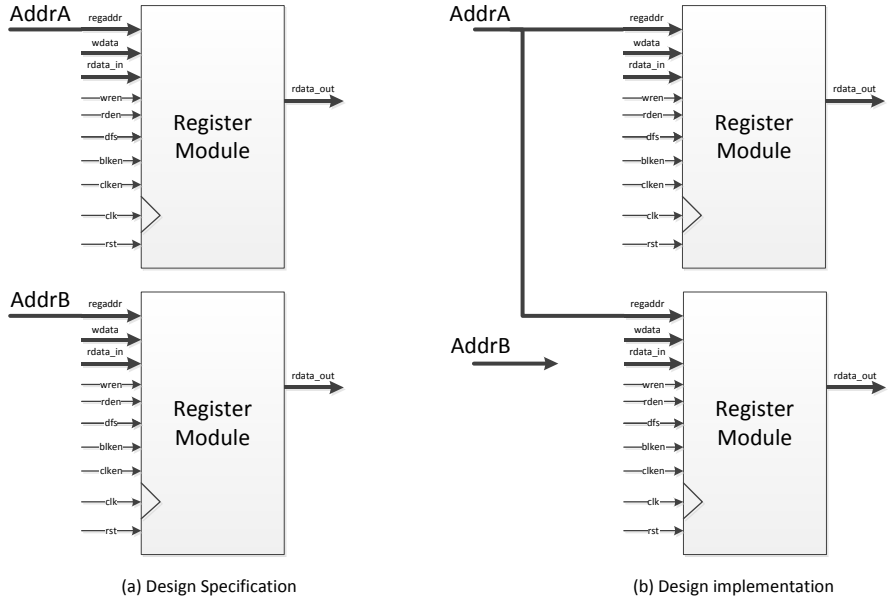

Figure 6. Register Aliasing issue due to address decoding error

B. Duplicated registers bugs in the design

Another bug we found in our design is that some legacy registers (address space) from the old projects have been used to implement new registers and design logic in the new project. However, the old design logic was not removed completely from the design due to the copy/paste/port-over procedure for faster implementation. The fanouts of the legacy register were connected to the logic that affected the status/value of some other real registers in the design. Thus changing the value of the legacy register changes the value of other registers unintentionally, resulting in the wrong behavior. This is illustrated in Figure 7. Legacy register RegA shared the same address with real RegB. When RegB is written with some value, RegA is also written with the same value. The value would have affected the fanout logic RegA connected to, and eventually the status of RegC. This type of register aliasing issue is hard to identify by either the traditional register aliasing checking sequence or by the three sequences discussed in section IV. But the issue is easily captured by our read/write_enable checking models.

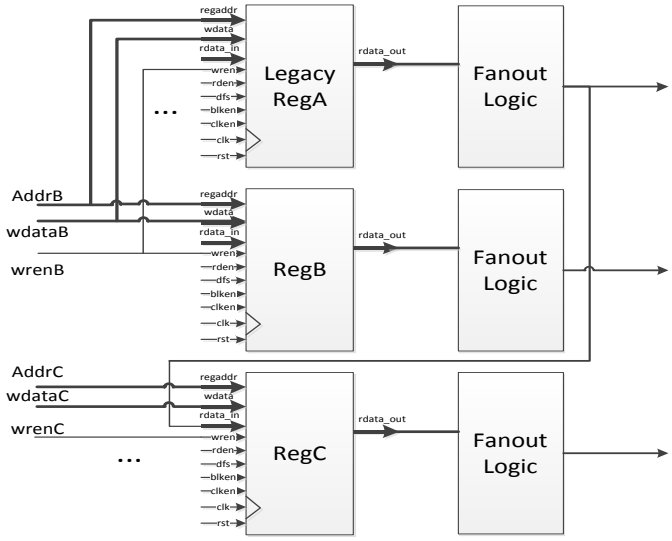

Figure 7. Register aliasing issue due to the duplicated Registers

C. Multiple registers read/write bugs in the design

The last type of bug we found in our design was due to the logic error of the read_enable and write_enable control signals. In one of our projects, we have two register blocks that share the same read_enable and write_enable signals. Another block_select signal was used to select which block should be enabled for access. The example code is shown in Figure 8. The blk_sel signal didn't expect to change during the duration when rden was asserted. However, due to the design bug, blk_sel changed in the middle of rden being asserted. blk2_rden was asserted right after blk1_rden was asserted. This caused a false read to a register in block 2. If this register were a clear-after-read register, the status of that register would be cleared by this false read. A similar scenario would happen to writes too if the write logic were wrong. The generated false writes would change the register value. The example waveform is shown in Figure 9.

```
1   reg rden, blk_sel;
2   wire blk1_rden, blk2_rden;
3   …
4   blk1_rden = rden & blk_sel;
5   blk2_rden = rden & ~blk_sel;
```

Figure 8. Example code for multiple register read/write bug

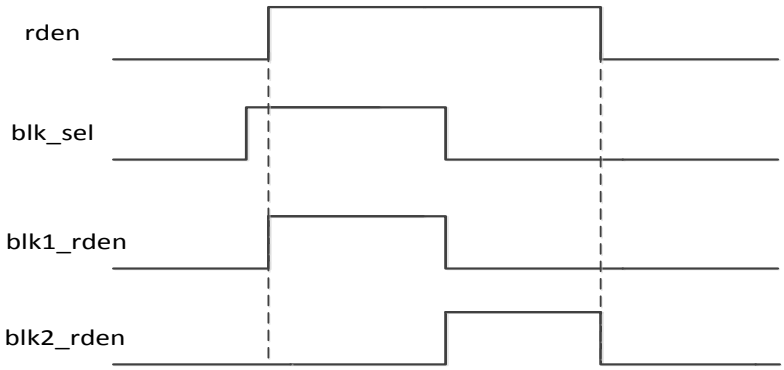

Figure 9. Waveform for multiple register read/write bug

This type of error can be captured only by using our read_enable and write_enable checking model. As presented in section III, the counters' checking not only checks that each register access triggers only one register access, but they also check that when there is no register access expected, no register access happens. The checking for the latter part would fail for this type of error because the counters would be cleared at the posedge of reg_rd_chk_on/reg_wr_chk_on and not at the negedge of reg_rd_chk_on/reg_wr_chk_on. This guarantees that before the next register access, no more than one register access would happen. In our case, two register reads would happen for one register access, the read counter would be counted to 2, thus causing the assertion to fail.

Our methodology can also be extended to handle broadcast and multicast register accesses. For broadcast and multicast registers, we need to identify the number of parallel accesses to them and adjust the assertions to check counter values for those registers accordingly.

Additionally, the performance has significantly improved by using our methodology. In our projects, we have thousands of registers in a single chip. With this number of registers, it's impossible to run the traditional register aliasing check sequence on the whole chip domain. For example, with a simple calculation, for a chip with 3700 registers, it takes 24832 hours (1034 days) to check all the registers. So we have to split them into blocks and run the traditional register aliasing check sequence in the per-block domain. However, in this way we can't guarantee that we can find all the register aliasing bugs because if the register aliasing bug is across multiple blocks, we can't find it. To do the comparison, we tried to keep registers in the same blocks as much as possible when runtime allowed. For our methodology, the registers can be grouped into very fine granularity blocks. It doesn't affect its ability to find register aliasing bugs. With each test running on a single register block and multiple tests running in parallel on the LSF farm, all the tests can finish within hours. Table I shows the performance comparison of using the traditional register testing sequences vs. our sequences. We tried to keep the number of registers in one block to less

than 300 for the traditional method to work. The runtime varied for different interfaces to access registers. On average, 31.5x to 61x improvements can be achieved for each project. The bigger the register block, the larger the improvement due to the $O(n^2)$ complexity of the traditional method vs. the O(n) complexity of our methodology.

TABLE I

PERFORMANCE COMPARISON

Projects	Average num. of registers in one block	Average runtime of traditional register aliasing test sequence (hours)	Average runtime of register enable signal checking sequence (hours)	Improvements ($Runtime_{traditional}/Runtime_{new}$)
Project A	126	53.55h	1.70h	31.5x
Project B	192	66.86h	1.45h	46.1x
Project C	244	108h	1.77h	61x

VI. SUMMARY

In summary, the register aliasing issue is very hard to capture for register verification with today's complex ASIC designs. The traditional simulation-based verification method for finding the register aliasing issue is very time consuming, and, if not carefully designed, it cannot achieve the coverage completeness. We propose a register verification methodology to easily detect register aliasing issues. Our register read/write enable signal checking models and three register access sequences can achieve register verification coverage completeness. By applying this methodology to three projects within our high-speed interconnect physical layer products, several register aliasing bugs were found in the earlier stage of the design, giving us the confidence for a bug-free design and tapeout. Additionally, performance has significantly improved by using our methodology. 31.5x to 61x performance improvements were achieved over the traditional register testing sequences.

ACKNOWLEDGMENTS

We would like to thank Jing Li and Kang Xiao for their support and encouragement in completing this paper.

REFERENCES

[1] https://en.wikipedia.org/wiki/Aliasing_(computing)#Hardware_aliasing
[2] https://www.doulos.com/knowhow/sysverilog/ovm/tutorial_rgm_2/
[3] http://www.esacademy.com/en/library/technical-articles-and-documents/miscellaneous/software-based-memory-testing.html