Verifying RO registers: Challenges and the solution

Ivana Dobrilovic
Veriest Solutions
Belgrade, Serbia
ivanad@veriests.com

***Abstract*-The role of registers in hardware is to be a programming interface to the software. They provide the static information about the design, allow controlling the behavior or features of the design, and provide the ability for the software to figure out if the design is performing the operation as expected or if it needs some attention. Registers can be classified based on several criteria. Based on their function, they can be broadly divided into 3 categories: information, control, and status registers. There is also a division based on the type of access. The most common types are "RO" (read-only), "RW" (read-write) and "WO" (write-only). A read-only register is a special type of register that can only be read by the software. These registers are typically used for things such as status registers, certain control registers, hardware timers, and counters. Verification of a read-only register is typically tricky. This paper will describe the most common problems that can be encountered, as well as the solution.**

I. INTRODUCTION

Considering that hardware behavior is configurable and monitored through registers, it is not surprising that verifying them has become one of the primary tasks in chip development. And taking into account a large number of registers in today's designs and their dynamic nature, that is a very difficult and challenging process. Read-only registers, such as status registers, debug registers and counters, are especially problematic in this sense.

Basic implementation correctness (ability to access all the registers, correctness of all access types for different fields, the post-reset values matching the expectation) is typically verified by a single test exercising only that. The more challenging area is verifying the functional correctness of registers. For most of the registers, functional correctness is spread out across many tests. Those tests are focusing on the verification of a higher-level operation, but while doing that they use different registers. Hence, although it's not their main purpose, they verify registers indirectly.

Status registers, as an example of read-only registers, provide reports for various hardware events to the software. Some examples are interrupt status, error status, link operational status, FSM status, faults etc. Considering the importance of this functionality, the goal is to make sure these registers have the correct value at any time. The most convenient way to achieve that is to check their value occasionally. There are two aspects of the strategy for this verification task. The first one is related to the sequence. The recommendation is to add a sequence for registers polling that will run in parallel to the main test sequence. This sequence will initiate registers read transactions at random moments. The other aspect is the checking itself. That should be implemented in a common object that serves as a reference model, scoreboard, or some hierarchically lower instances. And actually, that is the challenging part.

II. CHALLENGES

In order to implement a checker for a specific register or a register field, one should have an expected register value reliably predicted. For a verification engineer that is sometimes hard to achieve if he considers DUT a black box as he is supposed to. There are cases when it is impossible to know the timings of some DUT internal events without looking at internal signals.

There are several problematic scenarios. All of them suggest that checkers have to be made more flexible and that there cannot be a single expected value, but rather a pool of them. The aim of this chapter is to explain those situations in more details and to give suggestions for each specific case how to add flexibility to the checkers.

A. Register value changes to fast

A problem that can be encountered with counter registers or some other statistic registers is that their value is changed too fast in relation to the bus transaction cycle.

B. Read cycle duration

Another issue that one can encounter trying to check register value at a random time is a huge time gap between the command and response phase on the interface itself. For instance, the DUT might be implemented to drive back read data as soon as it gets the command, but the verification testbench will get that data only after the response phase is completed (master accepted the read data). And the bus protocol might allow long response latency. During that gap, the expected register value might change several times and one cannot be sure which value would correspond to the actual bus transaction.

Figure 1 shows how *COUNTER_X* value changes during single read register transaction cycle. This counter does not even count on every clock cycle but on *CNT_PULSE*, whose frequency is much lower than the clock frequency. Still, it changes value two times while waiting on read response.

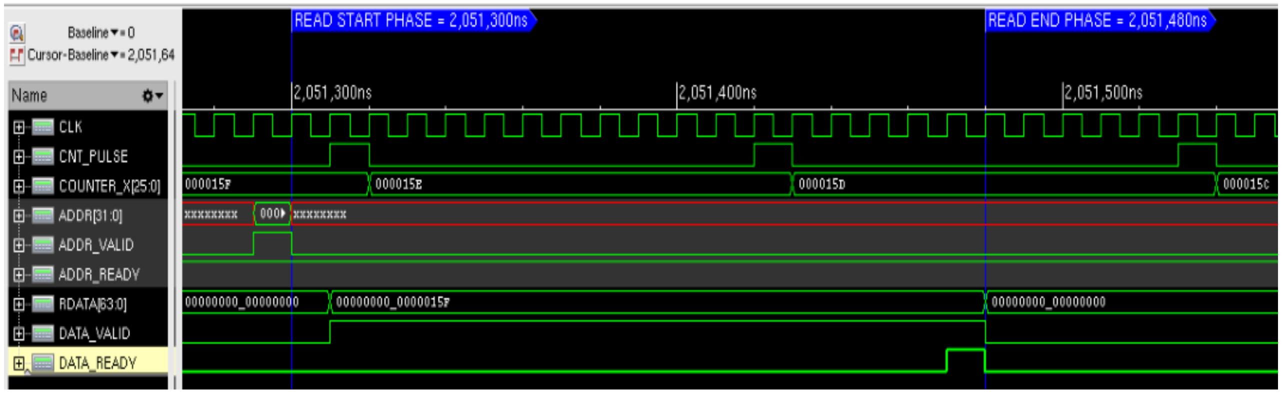

Figure 1. Counter values during one cycle of register read transaction.

C. Registers containing fields of various types

For registers fields that are 1 bit wide there is not room for any flexibility in checking since such field can have only one correct value.

Figure 2 is an example of how a 1-bit field (*STATUS_20*) of interrupt status register changes its value during a single register read cycle.

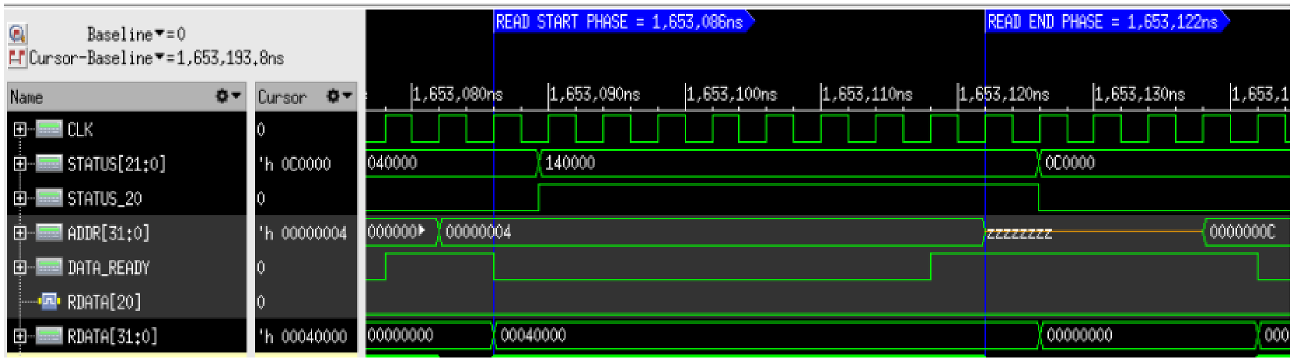

Figure 2. 1bit width status filed values during one cycle of register read transaction.

Also, there are status registers that contain fields of different access types and functionality. Value of each field in such register can be changed in the different state of the DUT i.e. on different events, state machine transitions, responses to outside requests, results of some calculations, etc. Therefore, these fields should be tested independently of each other.

D. Common race condition between ENV and DUT

Problem caused by long read response latency should be clear by now. However, even small latency can make race condition between expected and observed read value i.e. to make a prediction of the exact value impossible. That is explained using Figures 3 and 4. Value of *COUNTER_X* changes during the read transaction cycle even though response comes very quickly.

Figure 3 shows a non-problematic scenario. In this case, a value of *COUNTER_X* is changed at the beginning of the read transaction cycle which allows the read operation to catch the actual value of the counter (*COUNTER_X* and *READ_DATA* have the same value, ‘h24E).

Figure 4, on the other hand, represents how easily this can go wrong. Here the counter value is updated one clock cycle later and that results in read operation catching the old counter value (*READ_DATA* picks up the old *CONTER_X* value which is ‘h1DB, while the actual *COUNTER_X* value is ‘h1DA).

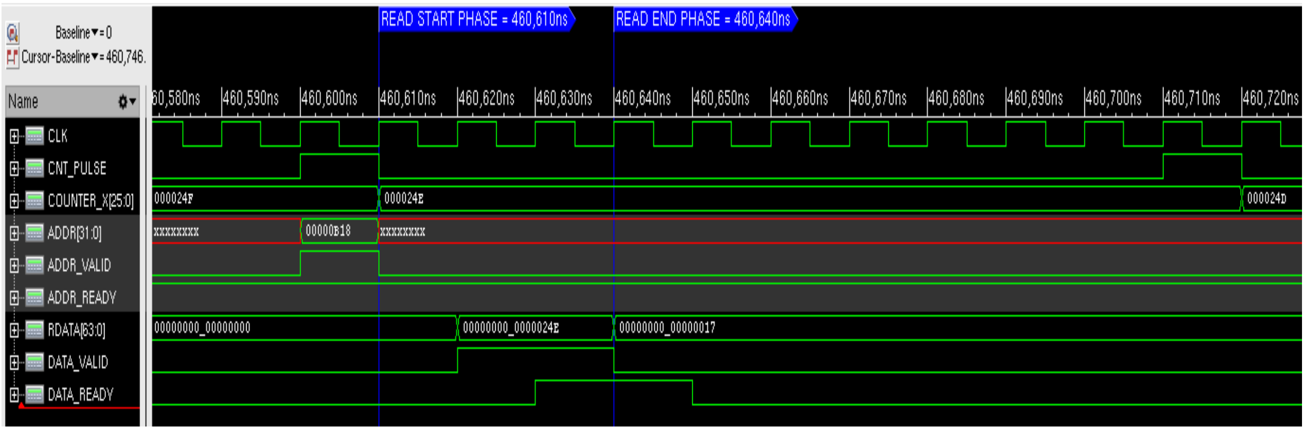

Figure 3. Counter values during one cycle of register read transaction.

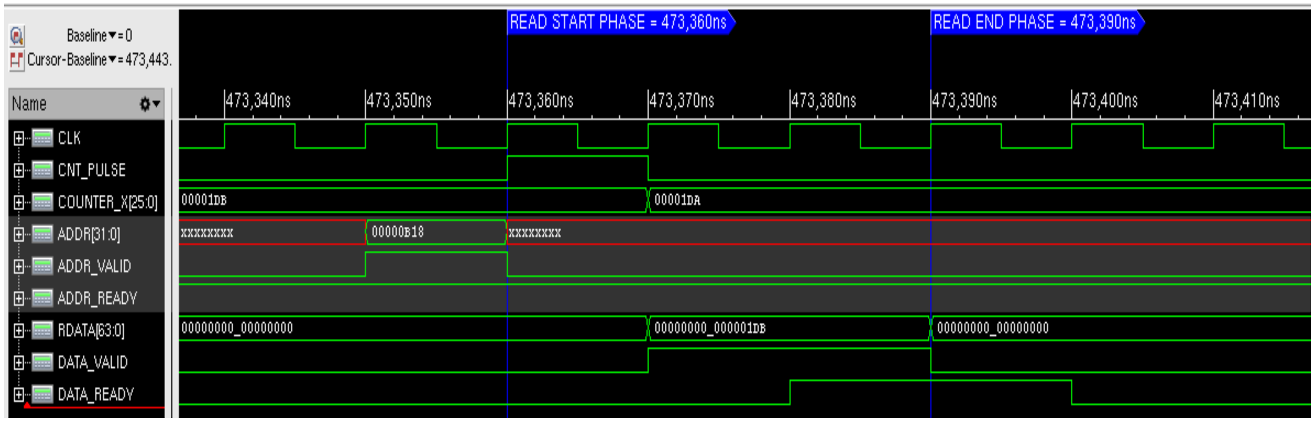

Figure 4. Counter values during one cycle of register read transaction.

III. SOLUTION IMPLEMENTATION

Since it is impossible to accurately predict expected value and create reliable checker based on it, a certain flexibility has to be added to the checker, as already mentioned. The best approach is to to determine the necessary and sufficient level of flexibility in the beginning. If you start from a strict checker and then try to introduce modifications, that will make the code less readable and hard to maintain.

This paper suggests a single solution that covers different read-only register types and most of the problematic situations in their verification. Solution relays on the fact that expected value of the register field could be only approximately predicted. Also, the solution takes care of all possible values and their transition order since that can be of a great importance.

The solution mechanism requires 3 variables per register or register field in the reference model:

• A field that will store an expected value,
• A flag that will signal if register reading is ongoing,
• A queue that will store all expected values created while the register reading was ongoing.

The idea is to collect all the expected register values during the read transaction targeting the specific register. Once the transaction is completed, the checker will verify if the actual register value matches any of the expected values from the queue.

There is an example represented through code pieces in the figures bellow that helps explaining the solution in more details. In the example *BLOCK1* is a registers block, *REG1* is a register and *StatusA* is one its field. The field *StatusA* is *FAW* bits wide.

Figure 5 shows the declaration of all variables needed for the solution implementation. The fields being declared are the following:

- *reg1_read_ongoing* - the flag that will signal if register reading is ongoing,

- *reg1_statusA* - the variable that will store an expected value,
- *reg1_statusA_1* - the queue that will store all expected values created while the register reading was ongoing.

```
class model extends uvm_component;


    ..................................
    //BLOCK1 declaration
    block1_regs BLOCK1;

    //Status register check fileds
    bit                         reg1_read_ongoing;
    bit [FAW-1:0]               reg1_statusA;
    bit [FAW-1:0]               reg1_statusA_q[$];
    ..................................


endclass
```

Figure 5. Variables declaration

The expected value is calucalted based on the DUT functionality. That calculation is here represented by a method *calculate_reg1_statusA()* that is implemented in the reference model and shown in the Figure 6. This method runs all the time in the reference model, in parallel with everything else. The calculation itself is triggered by an event defined in the specification for the specific register field. Once triggered it calculates the expected value for that register field (*valueA*) and afterwards stores that value in *reg1_statusA*, a variable that is part of our solution mechanism. If reading of that register is ongoing, it also pushes the predicted value to the queue *reg1_statusA_q*. As already explain, the queue is actually the pool of the expected values which enables flexibility in checking the register field value.

```
function model::calculate_reg1_statusA();
    // ... calculation ...


    ..................................
    // valueA is the calculated value
    // and it is assigned to expected reg1_statusA
    reg1_statusA  = valueA;

    // Putting calculated value in a pull of expected value
    // in case that read is ongoing.
    if (reg1_read_ongoing == 1)begin
        reg1_statusA_q.push_back(reg1_statusA);
    end
endfunction
```

Figure 6. calculate_reg1_statusA

Registers are accessed using *protocol X*. Monitor of the *agent X* reports transaction using TLM analysis port. Implementation of the TLM analysis port *write()* method is done in the reference model and shown in Figure 7. Basically, once a start of a read operation for a specific register is detected, the flag which says that the read operation is ongoing is set and the expected value of the register field at that moment is pushed to the queue. This is the first expected value calculated during the read register cycle. For all other potential changes of the register field value that might happen until the end of the read operation, pushing to the queue will be done by *calculate_reg1_statusA()* method.

```
function model::write_X(X_seq_item tr);
        ...................................
        if(tr.phase == START && tr.direction == READ ) begin
        //Read operation started
        //Start collecting expected possible values for read data
        if (tr.address  == BLOCK1.REG1.get_address() ) begin
            reg1_read_ongoing = 1;
            reg1_statusA_q.push_back(reg1_statusA);
        end
endclass
```

Figure 7. X Write method implementation on starting phase

When *X monitor* reports the end phase of the transaction, it provides full information about that transaction, including the read data. Usually, the read data would be checked against the single expected data, but the solution from this paper suggests checking if the actual read data matches any element in the queue of expected values *reg1_statusA_q()*. If there is a match, the functionality is considered correct. If not, the reference model issues an error indication. This is shown in Figure 8.

```
function model::write_X(X_seq_item tr);
    ...................................
    if(tr.phase == END && tr.direction == READ ) begin
        //Read operation ended
        //Compare the read data against the queue elements
        if (tr.address  == BLOCK1.REG1.get_address() ) begin
            //check REG1 field
            if(tr.rdata inside {reg1_statusA_q})begin
                `uvm_info("reg1_statusA_check",
                         $sformatf("REG1 read:
                                    Observed rdata has correct value"),UVM_LOW);
            end else begin
                `uvm_info("reg1_statusA_check",
                         $sformatf("REG1 read - Observed mismatch:
                                   read data %0x, expected values %p",
                                   tr.rdata, reg1_statusA_q));
            end

            reg1_read_ongoing = 0;
            reg1_statusA_q.delete();
        end
    end
endfunction
```

Figure 8. X Write method implementation on ending phase

The same approach should be taken when there are multiple fields in the register. That is shown in Figure 9. Fields can have different trigger event, width, access policy.

```
class model extends uvm_component;
    ...................................
    //BLOCK1 declaration
    block1_regs BLOCK1;

    //Status register check fileds
    bit                     reg1_read_ongoing;
    bit [FAW-1:0]           reg1_statusA;
    bit [FAW-1:0]           reg1_statusA_q[$];

    bit [FBW-1:0]           reg1_statusB;
    bit [FBW-1:0]           reg1_statusB_q[$];
    bit                     reg1_statusC;
    bit                     reg1_statusC_q[$];
    ...................................
endclass
```

Figure 9. Multiple number of fields under check

In this case, *write_X()* function implementation looks like in Figure 10.

```
function model::write_X(X_seq_item tr);
    ...................................
    if(tr.phase == START && tr.direction == READ ) begin
        //Read just started
        //Start collecting expected possible values of read data
        if (tr.address  == BLOCK1.REG1.get_address() ) begin
            reg1_read_ongoing = 1;
            reg1_statusA_q.push_back(reg1_statusA);
            reg1_statusB_q.push_back(reg1_statusB);
            reg1_statusC_q.push_back(reg1_statusC);
        end
    end
    ...................................
    if(tr.phase == END && tr.direction == READ ) begin
        //Read just started
        //Start collecting expected possible values for read data
        if (tr.address  == BLOCK1.REG1.get_address() ) begin
            //check REG1 field
            if(tr.rdata inside {reg1_statusA_q})begin
                `uvm_info("reg1_statusA_check",
                          $sformatf("REG1 read:
                                     Observed rdata has correct value"),UVM_LOW);
            end else begin
```

```
                `uvm_info("reg1_statusA_check",
                         $sformatf("REG1 read - Observed mismatch:
                                    read data %0x, expected values %p",
                                    tr.rdata, reg1_statusA_q));
            end
            if(tr.rdata inside {reg1_statusB_q})begin
                `uvm_info("reg1_statusB_check",
                          $sformatf("REG1 read:
                                      Observed rdata has correct value"),UVM_LOW);
            end else begin
                `uvm_info("reg1_statusB_check",
                         $sformatf("REG1 read - Observed mismatch:
                                    read data %0x, expected values %p",
                                    tr.rdata, reg1_statusB_q));
            end
            if(tr.rdata inside {reg1_statusC_q})begin
                `uvm_info("reg1_statusC_check",
                          $sformatf("REG1 read:
                                      Observed rdata has correct value"),UVM_LOW);
            end else begin
                `uvm_info("reg1_statusC_check",
                         $sformatf("REG1 read - Observed mismatch:
                                    read data %0x, expected values %p",
                                    tr.rdata, reg1_statusC_q));
            end

            reg1_read_ongoing = 0;
            reg1_statusA_q.delete();
            reg1_statusB_q.delete();
            reg1_statusC_q.delete();
        end
    end
endfunction
```

Figure 10. Multiple fields under check

As those fields can change their value on different trigger events, prediction of their values should be done separately in reference model. That can be seen in Figure 11.

```
function model::calculate_reg1_statusA();
    ...................................
    // valueA is the calculated value and it is assigned to expected reg1_statusA
    reg1_statusA  = valueA;

    // Putting calculated value in a pull of expected value
    // in case that read is ongoing.
    if (reg1_read_ongoing == 1)begin
```

```
        reg1_statusA_q.push_back(reg1_statusA);
    end
endfunction

function model::calculate_reg1_statusB();
    ...................................
    // valueB is the calculated value
    // and it is assigned to expected reg1_statusB
    reg1_statusB  = valueB;
    // Putting calculated value in a pull of expected value
    // in case that read is ongoing.
    if (reg1_read_ongoing == 1)begin
        reg1_filedB_q.push_back(reg1_statusB);
    end
endfunction

function model::calculate_reg1_statusB();
    ...................................
    // valueC is the calculated value
    // and it is assigned to expected reg1_statusC
    reg1_statusC  = valueC;
    // Putting calculated value in a pull of expected value
    // in case that read is ongoing.
    if (reg1_read_ongoing == 1)begin
        reg1_filedC_q.push_back(reg1_statusC);
    end
endfunction
```

Figure 11. Fields prediction methods

In addition to suggested solution, it is very important to implement coverage point where only one expected value was in the expected queue to prove that exact register/field value was checked, as shown in Figure 10.

```
function model::write_X(X_seq_item tr);
    ...................................
    if(tr.phase == END && tr.direction == READ ) begin
        //Read just started
        //Start collecting expected possible values for read data
        if (tr.address  == BLOCK1.REG1.get_address() ) begin
            //check REG1 field
            if(tr.rdata inside {reg1_statusA_q})begin
                `uvm_info("reg1_statusA_check",
                          $sformatf("REG1 read:
                                     Observed rdata has correct value"),UVM_LOW);
            end
            else begin
                `uvm_info("reg1_statusA_check",
                        $sformatf("REG1 read - Observed mismatch:
                                  read data %0x, expected values %p",
                                  tr.rdata, reg1_statusA_q));
            end

            model_cov.reg1_statusA_cg.sample(reg1_statusA_q.size());

            reg1_read_ongoing = 0;
            reg1_statusA_q.delete();
        end
    end
endfunction
```

Figure 11. Checks with reference to coverage

Speaking of coverage, *reg1_statusA_cg* cover group is implemented to cover size of the *reg1_statusA_q* distinguishing queue size of 1.

IV. CONCLUSIONS

Today almost every block in an electronic design contains a series of registers. On top of that, register functionality evolves throughout the project execution. Considering the total number of registers and their dynamic nature, verifying them became one of the most challenging tasks in chip development process. This paper is inspired by many issues encountered in practice. It aimed to provide comprehensive descriptions of all problematic situations, as well as detailed explanation of the suggested solution. Implementation of the complete flow is done using System Verilog, UVM methodology with regards to the UVM register model [1].

REFERENCES

[1] IEEE 1800.2-2020 Standard for Universal Verification Methodology