Register Verification: Do We Have Reliable Specification?

NamDo Kim, Samsung Electronics Co., Ltd., nd.kim@samsung.com
JunHyuk Park, Samsung Electronics Co., Ltd., jh23.park@samsung.com
Byeong Min, Samsung Electronics Co., Ltd., byeong.min@samsung.com
Wesley Park, Mentor Graphics Corporation, wesley_park@mentor.com

Abstract

We propose a new register verification method that leverages formal verification to automatically generate a complete access policy specification for IP memory-mapped registers. While traditional register verification uses simulation to check IP compliance with a manually written specification, our method uses formal verification to automatically generate an IP compliant specification that designers manually check against their design intent. We introduce a register model that overcomes limitations in the expressiveness of the predefined UVM and IP-XACT access polices. And, we present results from the successful application of our method to the register verification of three industrial designs.

1. INTRODUCTION

IP designs contain many externally accessible registers, referred to as memory-mapped registers, that are used to control behavior, check status, and transmit and receive data. The implementation complexity of memory-mapped registers is relatively low compared with other design elements. Hence, register verification, the verification of memory-mapped registers, is mostly done by directed or lightly randomized simulation test benches and is viewed as a tedious job.

Accessing memory-mapped registers is often no different from accessing general-purpose memory. Techniques used to verify general-purpose memory are a good starting point for register verification. However, to improve efficiency, memory-mapped registers can have complex access policies. For example, a read-to-clear policy automatically clears the content of a register after reading it. The contents of a memory-mapped register may also be modified by internal operations. These complicate the task of register verification.

Another practical difficulty is the lack of a reliable specification. The specification documents not uncommonly omit details, such as precisely when a memory-mapped register can be modified by internal operations. Developing a quality verification environment from an insufficient specification is a painful task: designers and verification engineers will be frustrated by many test failures resulting from undocumented behavior. To complete the verification task, they may mask or ignore failures. In the best scenario, the verification environment will be tediously updated until no more failures are observed.

In this paper, we use formal verification to overcome the practical challenges of producing a reliable specification. We start register verification assuming no specification is available. Instead of verifying a target design based on a given specification, we use formal verification to explore all possible behaviors for each memory-mapped register. To embrace the wide-range of register behaviors, we developed a register model based on four aspects of register behavior. A small program post-processes the formal verification results and determines how each register behaves. Then, it identifies the access policies consistent with that behavior. If the identification matches the designer’s intention, the verification is already done, since formal verification has thoroughly verified all possible behaviors. If it doesn’t match, then there is either a design bug or an overlooked aspect of the registers’ behavior.

In Section 2, we give an overview and describe the benefits of our proposed register verification method. The implementation is introduced at a high-level in Section 3. In Section 4, we show a few different approaches to developing properties for formal verification, and discuss the implementation and the instantiation of the properties. The post-processing required after formal verification is described in Section 5. We present results in Section 6 and conclude in Section 7.

2. OVERVIEW

Figure 1 shows the steps in our verification flow. A target design is all that is needed to start the register verification; no specification of the memory-mapped registers in the design is required. The *Register Verification Package* (RVP) binds a set of properties that explores the register behavior of the target design. We call this type of property an *exploration property*. Formal verification exhaustively explores the design and determines which exploration properties are satisfied.

For the post-processing step, we developed a small program to interpret the formal verification results and produce comprehensive information on register behavior in a form of high-level specification. Because formal verification is exhaustive, the information produced in this step reflects the implementation of the individual memory-mapped registers in the design. We call this the *implemented* specification. If the designer sees that the implemented specification is exactly aligned with his design intent, register verification is complete. The implemented specification can be treated as the concrete *intended* specification. If designer sees a mismatch, it is either a design bug or an overlooked aspect of the register behavior.

The ability to detect overlooked register behavior is a key benefit of our method. It is difficult to manually maintain a good quality specification because of frequent design changes and complicated register behaviors. When verification is performed using an incomplete or unreliable specification, bugs are much more likely to escape to silicon. Our method leverages the exhaustive analysis of formal verification to avoid this common pitfall.

3. HIGH-LEVEL IMPLEMENTATION

To easily expand support to a variety of interfaces, we structured the implementation of our RVP into two layers: Bus Layer and Register Layer, as depicted in Figure 2.

Bus Layer directly interfaces with the target design and converts the interface protocol of the design into a standard interface we use for Register Layer. This isolates the Register Layer from the design interface. For a design with an AMBA (Advanced Microcontroller Bus Architecture) APB (Advanced Peripheral Bus) [1] interface for memory-mapped register access, Bus Layer translates the APB transactions and generates corresponding Register Layer interface signals. Figure 3 shows the interface of RVP for AMBA APB interface and an example of binding it to a design using SystemVerilog 'bind' command.

The exploration properties are located in Register Layer. The interface protocol of Register Layer is designed to enable efficient modeling of exploration properties. The exploration properties are designed to explore the behavior of a single bit of a memory-mapped register. Hence, Register Layer is instantiated for every individual bit of a target design.

The Register Layer interface is depicted in Figure 4. The ports are all input ports because they monitor the transactions of the target design and are used for implementing the exploration properties. 'write' is high only at the clock cycle that a write transaction from the interface performs a write operation on a targeted bit of the register. 'write_data' contains the data to be written when 'write' is high. For read transactions and data, there are similarly 'read' and 'read_data' ports. These four ports are the basic signals used in the exploration properties. As will be discussed in Sections 4.2 and 4.6, a few additional interface ports also need to be added, depending on the observation method.

4. REGISTER LAYER DESIGN

This section describes the exploration properties we use with formal verification to identify register behavior. The designs used in our experiments were thought to have simple memory-mapped registers and had been verified under UVM (Universal Verification Methodology) environments. UVM has well defined register verification methodologies [2] and our approach starts from exploring register behavior based on the UVM register access policies. To overcome limitations in these predefined access policies, we created a register model for specifying register behavior that is automatically extracted by a program. We call the register model *Atomic Register Behavior Model* (ARBM) and explain it in Section 4.3.

4.1 UVM-based Front-door Implementation

UVM has total 25 predefined access policies for commonly designed memory-mapped registers. For example, it defines RW access policy for normal read and write access. To verify whether a write operation updates the value to the expected value, the value stored in a target register must be observed. Front-door access uses the same bus interface for observing operations. This is commonly used in simulation-based register verification because it does not require an additional interface for observation. In contrast, back-door access uses a different interface for observation -- usually direct access to registers. Back-door access enables efficient observation, but requires the use of an additional interface.

To implement a property that uses front-door access, two steps are required. The first step is to initiate a bus transaction that sets the value of a target register to an expected value. The next step is to read the target register value using another bus transaction. By comparing the read value to the expected value of the first step, a property can verify whether the bus transaction in the first step actually set the target register to the expected value. Between these steps, any number of operations or idle cycles are allowed, except of course bus transactions that alter the register value. This enables complete verification of a given access policy.

Figure 5 shows an example property that verifies RW access policy. The assert property, 'a_access_policy_RW', corresponds to the second step explained above. 'nd_written' is a wire in the modeling layer that indicates whether the first step has been completed: a specific write operation to a target register. It remains high once asserted. Since formal verification is exhaustive and the specific write operation is non-deterministically selected, all possible write operations are analyzed. 'read' signal becomes high only at the cycle when a read operation is applied to the target register and the read value is available for verification. 'read_data == written_data' compares the read data to the expected value. Although this property is designed to verify RW access policy for one write operation at a time, non-determinism in conjunction with formal verification extends it to cover all possible write operations. 'c_write' is a constraint property that disallows any subsequent bus transactions that would alter the register value: once a write operation of interest occurs, there must be no other write operations that can alter the previously written value before it is observed in the second step.

Front-door implementation is commonly used for register verification because all the control and observation operations go through the same bus interface. However, it has inherent limitations. Access policies such as WO have no way to read back the written value from a register because read operations have no effect and cannot observe the register value. Likewise, RO access policy cannot determine whether the read value is correct because write operations have no effect and cannot set expected values to the register.

4.2 UVM-based Back-door Implementation

The limitations of front-door implementation can be addressed with direct access to a register. With direct access, called back-door access, observation can be done immediately by reading the value directly from a register. WO access policy can be verified by directly reading values using back-door access after write operations, without relying on read operations over the bus interface to verify the written values. RO access policy also can be verified by reading the expected values directly via back-door. Furthermore, back-door access produces simpler properties. Compared to back-door access, verifying an access policy using front-door access requires an extra bus transaction to observe the value of a register.

Back-door access requires the hierarchical HDL path to the register. Adding an additional port, 'register', to the Register Layer and connecting it to the HDL path of a register, is sufficient to enable back-door access to the register. Figure 6 shows examples of exploration properties implemented using back-door access. 'a_read' property verifies if a read transaction reads the current value of a register. As shown in the example, when 'read' occurs, the property compares the read data, 'read_data', to the current register value, 'register', directly from the register itself. Likewise, 'a_write' property verifies whether the write data is written to a register by comparing 'write_data' to 'register'. Here it is implemented as comparing the previous value of 'write_data' because by definition the register value is updated one cycle after a write transaction.

As shown in Figure 6, the properties use no bus transactions other than the bus transaction being verified. This simplifies debugging for design and verification engineers when formal verification produces a counterexample. The counterexamples are more compact than counterexamples generated by front-door properties.

4.3 Atomic Register Behavior Model

The predefined access policies in the UVM register model do not reflect all the register behaviors that can exist in a design. IP-XACT predefines more behaviors than UVM, but is still limited. Both UVM and IP_XACT allow user-defined extensions [3] to work around these limitations. But, it is also important to report coverage information, such as whether both 0 and 1 values can be written to a register or just one value is allowed. This kind of coverage information is not represented well using UVM or IP_XACT. We found the need for a more concise and complete specification format than UVM and IP_XACT offered. Hence, we defined our own register model, ARBM.

ARBM splits register behavior into four components: bus transaction type, bus transaction data, register value before bus transaction, and register value after bus transaction. If a register can be modified only through the external interface, the effects of transactions are immediate, and each bit behaves independently, then all the behaviors can be compactly represented. For bus interfaces that have only read and write operations, sixteen combinations of the four components are possible: {2 bus transaction types: read, write} x {2 bus transaction data values: 0, 1} x {2 register values before bus transactions} x {2 register values after bus transactions}.

When appropriate, we use symbols to express the relationship between components.

1. A symbol represents the possibility of both 0 and 1 values. If a number is used instead of a symbol, the component covers only that specific value.
2. If the same symbol is used in multiple components, the components are related to each other and have the same value.
3. If a symbol appears as an uppercase letter, the component is related, but has the opposite value of the symbol appearing as a lowercase letter.

Based on the above symbol assignment rules, we define *atomic access policies* using a 5-character format that expresses the possible register behaviors based on relationships between the components.

1. The first character represents the bus transaction type. The letter is 'R' if a bus transaction is to read from a register. 'W' if it is to write to a register.
2. The second character represents the bus transaction data. It can be 0, 1, or a symbol.
3. For readability, the third character is '_'.
4. The fourth character represents the register value before bus transaction. It can be 0, 1 or a symbol.
5. The fifth character represents the register value after bus transaction. It can be 0, 1 or a symbol.

For example, 'Rx_xx' represents a normal read: let the register value before a read transaction be 'x'; the read data will be 'x' and the register value after the read transaction will also be 'x'. Since the symbol 'x' covers both values, 0 and 1, the register is capable of storing any value. In the same manner, 'Wy_xy' represents a register with normal write: let the register value before a write transaction be 'x' and write data is 'y'; the register value after the write transaction will be 'y'. This implies that write data is independent to the previous register value; the register will be written with the write data regardless of the register value before the write transaction. For a complete description of register behavior, we use '+' symbol to combine multiple atomic access policies. This creates *ARBM access policies*. For example, 'Rx_xx + Wy_xy' access policy represents a register that complies with UVM's RW access policy.

The implementation of exploration properties for ARBM access policies uses back-door access and is similar to UVM-based back-door implementation. Figure 7 shows properties for R0_00 and W0_00. We do not generate properties for access policies with symbols. Instead, we synthesize access policies with symbols by post-processing the results of properties for access policies without symbols. For example, Rx_xx is implied when the sanity checks [4] of the properties for R0_00 and R1_11 access policies are covered and the sanity checks for R0_01, R0_10, R0_11, R1_00, R1_01, and R1_10 are provably uncoverable. By exploiting the sanity checks, we reduced the number of properties required from 48, one per atomic access policy, to 8.

Another benefit of using sanity results is coverage information. When formal verification directly proves properties for access policies with symbols such as Rx_xx, we cannot know whether both 0 and 1 are covered; perhaps read operations always return 1. However, if both sanity of R0_00 and R1_11 are proven to be coverable, then we can clearly represent that the register follows Rx_xx access policy.

4.4 Modifier Indicators

In Section 4.3, we defined ARBM access policies based on three assumptions: a register is modified only through the external interface, the effects of transactions are immediate, and each bit behaves independently. However, memory-mapped registers that violate these assumptions are not uncommon. Volatile registers [3] update their values without any external bus transaction to reflect the internal states of a design. To address any register behavior beyond the above assumption, we developed *modifier indicators* that represent which operation can alter the register value.

If a register can be modified when the assumptions are violated, the modifier indicator uses the letter U to represent the unidentified source of the modification. If a register is modified satisfying the assumptions, the modifier indicator represents the specific operation that alters the register value. R indicates that read transactions alter the register value. W is used for write transactions, W0 for write with data value 0, and W1 for write with data value 1. If a register cannot be modified, the modifier indicator C represents the constant register value. If a register can be modified only once by the first operation after a global reset, the modifier indicator uses letter O. Figure 8 shows example properties for the modifier indicator R and O.

ARBM access policies are written with modifier indicators in front and explicitly represent which operation can modify the register value. Multiple modifier indicators can be combined. For example, 'RW + Rx_x0 + Wy_xy' represents an access policy similar to UVM's WRC access policy. However, 'RW' indicates that the register can only be modified by read or write transactions and 'Rx_x0 + Wy_xy' indicates that the register can have all possible values.

4.5 Reset Value Exploration

Register reset value verification is relatively simple. However, we found some cases where the actual reset value cannot be observed using front-door access; the register updates its value before any read bus transactions are possible. To identify this register behavior, we applied both back-door and front-door accesses for reset value exploration properties. Back-door access identifies the actual reset value. Front-door access reads the register value at the earliest possible bus transaction after reset. To represent these two values in access policies, we combine them with comma. For example, '0,X' indicates the actual reset value is 0 but the earliest possible read operation might read 1 instead of 0.

4.6 Local Reset Exploration

The reset value verification explained in Section 4.5 is focused on the global reset of a design. A register, however, can have other type of resets that are generally known as local resets. Different from global resets, local resets are applied during normal operations, possibly multiple times, to a subset of a design. Because of local resets, many register are identified as volatile. We isolate the local reset behavior from other register behaviors by developing exploration properties for local resets. The goals of these exploration properties are:

1. For each register, identify whether it is reset by a local reset. If it is, then also identify the reset value, the polarity of the local reset signal, and whether it is asynchronous or synchronous reset.
2. For a volatile register with a local reset, determine whether it can be a non-volatile register if the local reset remains inactive.

To observe local reset behavior, we added a back-door access port, 'local_reset', to the interface of Register Layer. Figure 9 shows the implementation of the local reset exploration properties. Using SystemVerilog generate statement, 4 different combinations of 'local_reset_polarity' and 'local_reset_async' values are applied to generate total 8 properties of 'a_local_reset_value_0' and 'a_local_reset_value_1'.

To achieve the second goal, the exploration properties for ARBM access policies are also modified. The 'disable iff' condition considers not only global reset but also local resets.

To describe local reset behavior in ARBM access policies, we use the format: 'V:AP:NAME'. 'NAME' indicates the name of the local reset. 'V' represents the reset value, 0 or 1. 'AP' is a two character code where the first character indicates whether the reset is 'A'synchronous or 'S'ynchronous, and the second character indicates the polarity, active 'H'igh or active 'L'ow.

4.7 Instantiating Properties

When the RVP is bounded to a target design, the Bus Layer is instantiated only once. However, the Register Layer needs to be instantiated multiple times by the Bus Layer, once for each register bit that is to be verified. For UVM-based front-door exploration properties, this instantiation is accomplished by SystemVerilog 'generate' statement. The necessary information: address range and bit width, is passed via parameters as shown in Figure 3.

However, the SystemVerilog 'generate' statement cannot be used for connecting back-door signals. For UVM-based back-door and ARBM exploration properties, we developed a small script that instantiates the Register Layers and connects back-door signals to the interface ports. This script reads relevant information from a file, called 'register information file'. As shown in Figure 10, it has a list of registers along with the address and bit information. Local reset signals can be included as well.

When a property is instantiated for a register, a special naming rule is used for post-processing. The instance of a property includes the address and bit information in its instance name. If local reset exploration is applied, the instance name also includes local reset name and values for polarity and asynchronous variables. After the property instances are verified by formal verification, post-processing reads the results and parses the instance names to recognize which property is verified with which register.

5. POST-PROCESSING

After formal verification analyzes the exploration properties developed in Section 4, a post-processing step examines the results for each register and determines the register's behavior. For the UVM-based properties, the post-processing program reports the identified behavior based on the predefined UVM access policies. For the ARBM properties, the report uses the ARBM access policies.

Because we developed properties to correspond to access policies, if a property is proven, it implies that the register always complies with the corresponding access policy. In many cases, more than one property is proven for a register. If two proven properties represent orthogonal access policies, both are reported; otherwise, post-processing selects the access policy that best represents the register. For example, a register with UVM RW access policy will have its 'a_read' and 'a_write' properties proven. Its 'a_W1S' and 'a_W0C' properties will also be proven, because normal writes will set the register value to 1 if the write data is 1 and clear it to 0 if the write data is 0. Post-processing distills these results into RW access policy, hiding the property details.

Figure 11 is a decision table that maps the property results to the most appropriate access policy. The post-processing program uses this decision table to report the UVM predefined access policies. If formal verification results do not match any entry in the decision table, a new entry is added. New entries are named UNDEF0, UNDEF1, UNDEF2, and so forth.

Post-processing the results of ARBM exploration properties starts by determining the global reset value for a register. From the instance name of a proven property, we can identify the reset value and the register. If no global reset exploration properties are proven, post-processing reports an 'X'; either the register is not reset by a global reset or the reset value is variable.

The next post-processing step determines whether a register has a local reset. For each local reset and register pair, 8 properties are instantiated. If one of the properties is proven, the post-processing program parses the instance name of that property to identify which local reset is applied, its polarity, whether it is asynchronous or synchronous, and the register reset value. Once the local reset is identified for a register, properties which do not have the local reset in 'disable iff' are ignored. This is necessary to isolate the effects of the local reset from other register behavior as explained in Section 4.6.

After identifying local resets and focusing on the appropriate set of exploration properties, the post-processing program identifies modifier indicators and ARBM access policies. The access policies are identified by applying the symbol assignment rules to the sanity check results of the properties for ARBM access policies.

6. RESULTS

We applied our register verification methodology to three different industrial designs, all of which are implemented in silicon. We used Questa Formal, an industrial model checking tool, to formally explore the register behavior with the exploration properties. These designs use AMBA APB interface for memory-mapped registers. We compare the register verification results from UVM-based back-door exploration and ARBM against the original specification documents.

Figure 12 shows the log of the ARBM post-processing program. 'read_rpt' command shows some brief statistics. 'read_reg_info' reads a register information file. 'verify_reg' analyzes the formal verification results and identifies each register's behavior. 'report_reg' reports the access policies in various formats. '-uniq' is an option to store only the registers with a unique behavior. We generate all the logs with '-uniq' option to review only the unique behaviors. '-verbose' is an option to display the sanity information along with the report.

As shown in the log, total 320 bits are verified in 10 addresses for design A. For each bit, 30 ARBM properties are instantiated to explore the register behavior: 4 for global reset, 2 for local reset, 8 for modifier, 8 for read operation, and 8 for write operation. 2 local reset properties are instantiated as safety-empty [5] in design A because design A has no local reset.

Figure 13 shows the log from the post-processing program for UVM-based back-door exploration. The third column represents the identified access policies. The extra columns are provided to show which properties are proven; these are useful when a predefined access policy cannot be identified (e.g. UNDEF0). RA0 and RA1 properties stand for Read Always 0 and 1, respectively. UBA stands for Updated By Access, which checks if only read or write operations can modify a register. These properties are added to help understand the behaviors of undefined access policies.

The register of address 0000 and bit 0 has normal read and write access behavior. UVM-based back-door and ARBM explorations identify it as RW and W + Rx_xx + Wy_xy, respectively. ARBM also reports other aspects of the register with the identified access policies. First, by reading the register information file, it can tell the name of the actual register in the design. If the actual register exists, R column indicates it as 'Y'. If it doesn't, 'N' will be used to indicate that the register is accessible by bus transactions but does not have an actual signal in the design. G_RST column shows access policies in terms of the global reset value. L_RST column shows access policies for local resets. Since design A doesn't have local resets, this column remains empty for all registers.

The register of address 0000 and bit 7 has UNDEF0 by UVM-based back-door exploration because the property results do not match any condition of the decision table. The proven properties are read, RC, WC, W1C, W0C, RA0, and UBA. What we can interpret from this is that no matter which bus transactions occurred before a read bus transaction, it always reads 0. Since the UVM predefined access policies do not have a specific entry for a constant register, the identification is resulted in UNDEF0. In contrast, ARBM exploration clearly shows this behavior in a comprehensive form, C + R0_00 + Wx_00: the read value is always 0 and the register remains constant 0.

The register of address 000c and bit 0 is the almost the same as the register of address 0000 and bit 0 except the reset value. The register of address 001c and bit 0 is identified as UNDEF1 by UVM-based back-door exploration. Since UBA is fired, we can interpret this one as a volatile register but it's very limited to understand further behavioral information. ARBM exploration is able to identify the behavior precisely. Rx_x0 indicates that read operations read the current register value correctly. But the register value is always cleared to 0 after read operations. 'x' indicates that both value 0 and 1 are covered. Wx_00 shows that any write bus transactions are ignored. By reviewing the design, we found that this identification is exactly aligned with the actual design intention: This register is a status register which updates its value at the read access, then the value is cleared to 0 until the next read access occurs. Because it updates the value at the read access, not after the read access, modifier is identified as 'U' instead of 'R'.

ARBM also reports the comprehensive behavioral information for the register of address 0000 and bit 0 in design B as shown in Figure 14. UVM-based exploration in Figure 15 has only two proven properties, read and UBA, and identify as RO. It is insufficient to understand all the behaviors and even misleading. However, ARBM identifies as W + Rx_xx + WX_xy + Wx_xx. While read is normal, write shows uncommon behavior: If the write data is the same as the register value, the register remains unchanged. If the write data is opposite to the register value, the register can have an uncorrelated value, that is, regardless of the write data, the register can be either 0 or 1. After taking a look at the design, we confirmed that ARBM exploration successfully discovered the correct design behavior: if the write data is opposite to the register value, the design conditionally accepts the data and, hence, the register value sometimes can change to the new value or remain the same.

The register of address 0000 and bit 8 has no actual register signal in the design. UVM-based exploration has only RA0 proven and mapped into UNDEF0. ARBM results in R0_xy + Wy_xz. Since there is no actual register, G_RST back-door value and Modifier have '–'. The only meaningful information here is the read data and G_RST front-door value, which are always 0.

The register of address 0030 and bit 0 is one of the common implementations for an interrupt status register. It reflects the pending status of an interrupt: when an interrupt is received, the register is set to 1. It remains 1 until write operation with data 1 comes to clear the register. R0_0x + R1_11 explains that read data is correct but the register value after read operation can change from 0 to 1 conditionally. Wx_0y + W0_11 + W1_1x reflects all the possible combinations for 'write 1 to clear' plus situations that the register value remains or changes to 1 when a new interrupt comes. UVM-based exploration has no row for this register. It is because only 'read' property is proven for this register and it is not unique any more.

Design C has one local reset, 'NRESET'. From L_RST column in Figure 16, we can clearly understand that 'NRESET' is effective only for specific registers and it always clears the corresponding registers to 0 by acting as an asynchronous and active low reset. Since ARBM exploration can isolate this local reset behavior from register behavior, we get W + Rx_xx + Wy_xy for registers like address 0000 and bit 1; the access policy implies that, if the effective local reset remain inactive, this register behaves as a normal read write register. If the local reset exploration is not applied, the identified access policy would have been U + Rx_xx + W0_x0 + W1_xy or anything with U.

For all three designs, we compared the results from our register verification methodology with the specification documents. We found small mistakes, such as a register field being mis-specified as [6:0] instead of [7:0]. But we also found many serious errors where the access policies are incompletely described, such as RO or RW when the actual behavior is more complex. We measured the number of mis-specified bits in the specification documents: out of the 992 bits verified in the three designs, 145 bits were mis-specified, or 14.6% of the total memory-mapped register bits.

7. CONCLUSION

Quality verification cannot be achieved without quality specification. However, it is very difficult to maintain a good quality specification. For register verification, we presented a new methodology that explores the register behavior using formal verification and post-processes the results to automatically generate a specification. We defined a set of comprehensive access policies based on ARBM and a concise specification format, which specifies the possible register values. We developed exploration properties for UVM-based front-door, UVM-based back-door and ARBM. We applied our methodology to three industrial designs. ARBM was able to identify complex behaviors that were not identified with the UVM-based property sets. With the ARBM access policies, it was possible to precisely describe all the behaviors. When we compared our automatically generated specification with the original, manually written design specification, we were able to identify many inaccuracies in the original documentation. By automatically generating an implemented specification from the design, verification is reduced to the straightforward task of checking that the specification matches the designer's intent.

8. REFERENCES

[1] AMBA™ Specification Rev 2.0, http://www.arm.com/

[2] Universal Verification Methodology (UVM) 1.1 User's Guide, http://www.uvmworld.org/

[3] IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows, http://standards.ieee.org/

[4] IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language, http://standards.ieee.org/

[5] Questa® Formal User Guide, http://www.mentor.com/

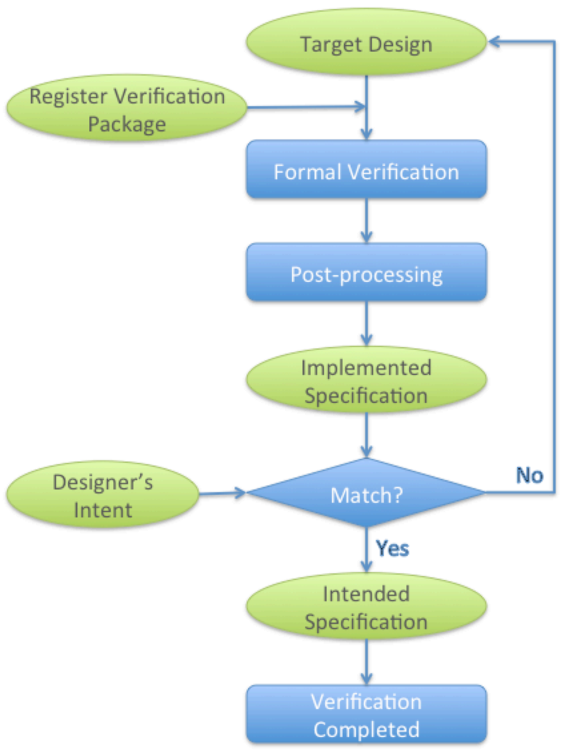


Figure 1. The flow of proposed register verification method

(Green ovals denote either input or output of a process. Blue rectangles denote processing steps.)

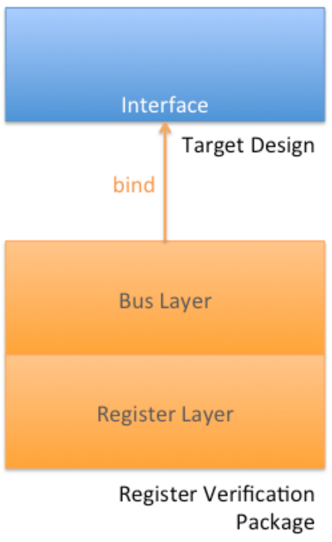


Figure 2. Block diagram of a target design and Register Verification Package for formal verification

```
module bindings;
    bind apb_design ww_reg_package_apb u_reg_package (.*);
endmodule

module ww_reg_package_apb #(
    parameter ADDR_BASE     = 32'h12c0_0000,
    parameter ADDR_TOP      = 32'h12c0_003c,
    parameter ADDR_WIDTH    = 32,
    parameter DATA_WIDTH    = 32
) (
    input                       PCLK,
    input                       PRESETn,
    input  [ADDR_WIDTH - 1:0]   PADDR,
    input                       PSEL,
    input                       PENABLE,
    input                       PWRITE,
    input  [DATA_WIDTH - 1:0]   PWDATA,
    input  [DATA_WIDTH - 1:0]   PRDATA
);
```

Figure 3. The interface of Register Verification Package for AMBA APB interface and an example of a bind statement

```
module ww_register_layer (
    input   write,
    input   write_data,
    input   read,
    input   read_data,

    input   clk,
    input   global_resetn
);
```

Figure 4. The interface of Register Layer

```
a_access_policy_RW: assert property (
    disable iff( ! resetn )
    nd_written && read
    |->
    read_data == written_data
);

c_write: assume property (
    disable iff( ! resetn )
    nd_written
    |->
    ! write
);
```

Figure 5. Example exploration properties of UVM-based front-door implementation

```
a_read: assert property (
    disable iff( ! resetn )
    read
    |->
    read_data == register
);
a_write: assert property (
    disable iff( ! resetn )
    write
    |->
    ##1 $past(write_data) == register
);
```

Figure 6. Example exploration properties of UVM-based back-door implementation

```
a_R0_00: assert property (
    (read && (register == 1'b0))
    |->
    (read_data == 1'b0)
    ##1
    (register == 1'b0)
);

a_W0_00: assert property (
    (write && (register == 1'b0))
    ##0
    (write_data == 1'b0)
    |->
    ##1
    (register == 1'b0)
);
```

Figure 7. Example properties for Atomic Register Behavior Model

```
a_changed_by_r: assert property (
    disable iff( ! resetn )
    $changed(register)
    |->
    $past(read)
);

a_changed_once: assert property (
    disable iff( ! resetn )
    register_changed
    |->
    ! $changed(register)
);
```

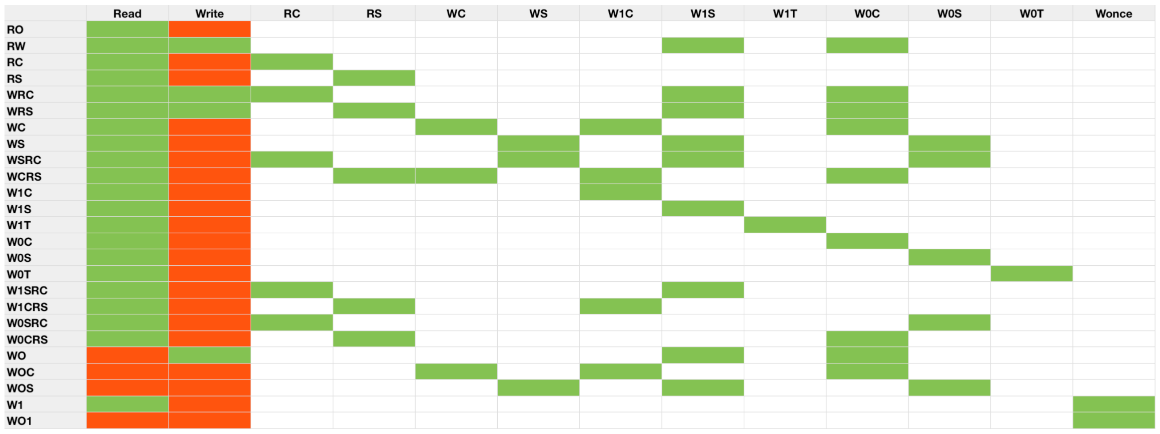
Figure 8. Example properties for modifier indicators

```
assign local_resetn
        = local_reset_polarity ?
        ! local_reset   : local_reset;
assign effective_local_resetn
        = local_reset_async     ?
          local_resetn : local_resetn_delayed;

a_local_reset_value_0: assert property (
    disable iff( ! global_resetn )
    (!effective_local_resetn)
    |->
    (register == 1'b0)
);

a_local_reset_value_1: assert property (
    disable iff( ! global_resetn )
    (!effective_local_resetn)
    |->
    (register == 1'b1)
);
```

Figure 9. Example properties for local reset exploration

```
# -------------------------------------------
# [ADDR] [BIT]  REG_PATH
# -------------------------------------------
[0000]   [31:0]   A.u0.CH_CFG[31:0]
[0004]   [31:0]   A.u0.Clk_CFG[31:0]
[0008]   [31:0]   A.u0.MODE_CFG[31:0]
[000c]   [31:0]   A.u0.SLAVE_SELECTION[31:0]

# -------------------------------------------
# SHORT_NAME LOCAL_RESET_PATH
# -------------------------------------------
NRESET  A.u0.NRESET
```

Figure 10. Example of a register information file



	Read	Write	RC	RS	WC	WS	W1C	W1S	W1T	W0C	W0S	W0T	Wonce
RO	🟩	🟥											
RW	🟩	🟩						🟩		🟩			
RC	🟩	🟥	🟩										
RS	🟩	🟥		🟩									
WRC	🟩	🟩	🟩					🟩		🟩			
WRS	🟩	🟩		🟩				🟩		🟩			
WC	🟩	🟥			🟩		🟩			🟩			
WS	🟩	🟥				🟩		🟩			🟩		
WSRC	🟩	🟥	🟩			🟩		🟩			🟩		
WCRS	🟩	🟥		🟩	🟩		🟩			🟩			
W1C	🟩	🟥					🟩						
W1S	🟩	🟥						🟩					
W1T	🟩	🟥							🟩				
W0C	🟩	🟥								🟩			
W0S	🟩	🟥									🟩		
W0T	🟩	🟥										🟩	
W1SRC	🟩	🟥	🟩					🟩					
W1CRS	🟩	🟥		🟩			🟩						
W0SRC	🟩	🟥	🟩								🟩		
W0CRS	🟩	🟥		🟩						🟩			
WO	🟥	🟩						🟩		🟩			
WOC	🟥	🟥			🟩		🟩			🟩			
WOS	🟥	🟥				🟩		🟩			🟩		
W1	🟩	🟥											🟩
WO1	🟥	🟥											🟩

Figure 11. A decision table to determine the best access policy from UVM-based back-door exploration results

(Columns correspond to each exploration property as conditions. Rows correspond to the UVM predefined access policies as determined policies. If a cross section is green, proof is expected for the condition property. If it is red, violation is expected. If it is white, the condition is don't-care.)

```
read_rpt ../Result.A A
# Reading files from '../Result.A' into version 'A'...
# Processing file '../Result.A/formal_verify.rpt'...
# TIME: Processing done at 01 sec: line 37943
# Property table is loaded: Assumptions (6)
# Property table is loaded: Active Targets (9600)
# Property table is loaded: Targets Proven (2152)
# Property table is loaded: Targets Vacuously Proven (3712)
# Property table is loaded: Targets Fired (3288)
# Property table is loaded: Targets Fired with Warnings (448)
# Property table is loaded: Target Waveforms (3736)
# Property table is loaded: Sanity Waveforms (2838)
read_reg_info ../Result.A/reg_info_file
# Reading register infos from '../Result.A/reg_info_file' into version 'A'...
# TIME: Processing done at 00 sec: line 16
# Total addr: 10
# Total  bit: 320
verify_reg -uniq
# Total register addr verified: 10
#                bits verified: 320
#                         READ: 320
#                        WRITE: 320
report_reg -verbose
# Address  Bit  Name                            R  G_RST  L_RST  Modifier  R-Policy  S0_00 S0_01 S1_00 S1_01  S0_10 S0_11 S1_10 S1_11  W-Policy  S0_00 S0_01 S1_00 S1_01  S0_10 S0_11 S1_10 S1_11
# -------  ---  ------------------------------  -  -----  -----  --------  --------  ----- ----- ----- -----  ----- ----- ----- -----  --------  ----- ----- ----- -----  ----- ----- ----- -----
#    0000    0  A.apb_if_u0.CH_CFG[0]           Y   0,0                W   Rx_xx       1                                          1   Wy_xy       1                 1      1                 1
#    0000    7  A.apb_if_u0.CH_CFG[7]           Y   0,0             C      R0_00       1                                              Wx_00       1           1
# -------  ---  ------------------------------  -  -----  -----  --------  --------  ----- ----- ----- -----  ----- ----- ----- -----  --------  ----- ----- ----- -----  ----- ----- ----- -----
#    000c    0  A.apb_if_u0.SLAVE_SELECTION[0]  Y   1,1                W   Rx_xx       1                                          1   Wy_xy       1                 1      1                 1
# -------  ---  ------------------------------  -  -----  -----  --------  --------  ----- ----- ----- -----  ----- ----- ----- -----  --------  ----- ----- ----- -----  ----- ----- ----- -----
#    001c    0  A.apb_if_u0.ReadData[0]         Y   0,0               ---U Rx_x0       1                                    1         Wx_00       1           1
report_reg
# Address  Bit  Name                            R  G_RST  L_RST  Modifier  R-Policy  W-Policy
# -------  ---  ------------------------------  -  -----  -----  --------  --------  --------
#    0000    0  A.apb_if_u0.CH_CFG[0]           Y   0,0                W   Rx_xx     Wy_xy
#    0000    7  A.apb_if_u0.CH_CFG[7]           Y   0,0             C      R0_00     Wx_00
# -------  ---  ------------------------------  -  -----  -----  --------  --------  --------
#    000c    0  A.apb_if_u0.SLAVE_SELECTION[0]  Y   1,1                W   Rx_xx     Wy_xy
# -------  ---  ------------------------------  -  -----  -----  --------  --------  --------
#    001c    0  A.apb_if_u0.ReadData[0]         Y   0,0               ---U Rx_x0     Wx_00
```

Figure 12. An example log generated by ARBM post-processing program with design A

```
# Address  Bit  Type     Read Write RC  RS  WC    WS W1C W1S W1T W0C   W0S W0T Wonce RA0 RA1   UBA
# -------  ---  -------  ----------------------  ----------------------  ----------------------  ----
# 0000       0  RW          P     P   F   F   F     F   F   P   F   P     F   F     F   F   F     P
# 0000       7  UNDEF0      P     F   P   F   P     F   P   F   F   P     F   F     F   P   F     P
# 001c       0  UNDEF1      P     F   P   F   P     F   P   F   F   P     F   F     F   F   F     F
```

Figure 13. An example log generated by the post-processing program for UVM-based back-door exploration with design A

```
# Address  Bit  Name                          R  G_RST  L_RST  Modifier  R-Policy     W-Policy
# -------  ---  ----------------------------- -  -----  -----  --------  -----------  -----------------
#    0000    0  B.b_interface.LCON[0]         Y   0,0               W    Rx_xx        WX_xy Wx_xx
#    0000    8  -                             N   -,0            ------  R0_xy        Wy_xz
# -------  ---  ----------------------------- -  -----  -----  --------  -----------  -----------------
#    0004    4  B.b_interface.UCON[4]         Y   0,0              ---U  Rx_xx        WX_xy Wx_xx
#    0004   12  B.b_interface.UCON[12]        Y   1,1               W    Rx_xx        WX_xy Wx_xx
# -------  ---  ----------------------------- -  -----  -----  --------  -----------  -----------------
#    0008    1  B.b_interface.FCON[1]         Y   0,0              ---U  R0_00        W0_00 W1_0x
# -------  ---  ----------------------------- -  -----  -----  --------  -----------  -----------------
#    0014    0  B.b_interface.ESTATUS[0]      Y   0,0              ---U  Rx_xy        Wy_xz
#    0014    4  B.b_interface.ESTATUS[4]      Y   0,0            C       R0_00        Wx_00
# -------  ---  ----------------------------- -  -----  -----  --------  -----------  -----------------
#    001c    1  B.b_interface.MSTATUS[1]      Y   X,X              ---U  Rx_xy        Wy_xz
# -------  ---  ----------------------------- -  -----  -----  --------  -----------  -----------------
#    0030    0  B.b_interface.INTP[0]         Y   0,0              ---U  R0_0x R1_11  Wx_0y W0_11 W1_1x
# -------  ---  ----------------------------- -  -----  -----  --------  -----------  -----------------
#    0034    2  B.b_interface.INTSP[2]        Y   0,0              ---U  R0_0x R1_11  Wx_0y Wx_11
# -------  ---  ----------------------------- -  -----  -----  --------  -----------  -----------------
#    003c    1  B.b_interface.VERSION_INFO[1] Y   1,1            C       R1_11        Wx_11
```

Figure 14. An example log generated by ARBM post-processing program with design B

```
# Address  Bit  Type     Read Write  RC  RS  WC    WS W1C W1S W1T W0C   W0S W0T Wonce RA0 RA1   UBA
# -------  ---  -------  ----------------------  --------------------  ----------------------  ----
# 0000       0  RO          P     F   F   F   F     F   F   F   F   F     F   F     F   F   F     P
# 0000       8  UNDEF0      F     F   F   F   F     F   F   F   F   F     F   F     F   P   F     F
# 0004       4  RO          P     F   F   F   F     F   F   F   F   F     F   F     F   F   F     F
# 0008       1  UNDEF1      P     F   P   F   F     F   F   F   F   P     F   F     F   P   F     F
# 0014       4  UNDEF2      P     F   P   F   P     F   P   F   F   P     F   F     F   P   F     P
# 003c       1  UNDEF3      P     F   F   P   F     P   F   P   F   F     P   F     F   F   P     P
```

Figure 15. An example log generated by the post-processing program for UVM-based back-door exploration with design B

```
# Address  Bit  Name                      R  G_RST  L_RST        Modifier  R-Policy     W-Policy
# -------  ---  ------------------------  -  -----  -----------  --------  -----------  -----------
#    0000    0  C.sfr.COLD_RESET          Y   0,0                    W     Rx_xx        Wy_xy
#    0000    1  C.sfr.Con_Ctl[0]          Y   0,0   0:AL:NRESET      W     Rx_xx        Wy_xy
#    0000    5  -                         N   -,0                 ------   R0_xy        Wy_xz
#    0000   24  C.sfr.Con_Ctl[17]         Y   0,0   0:AL:NRESET     ---U   R0_xx        Wy_xy
# -------  ---  ------------------------  -  -----  -----------  --------  -----------  -----------
#    0004    0  C.sfr.STATE[0]            Y   0,X   0:AL:NRESET     ---U   Rx_xy        Wy_xz
#    0004    1  C.sfr.STATE[1]            Y   0,0   0:AL:NRESET     ---U   Rx_xy        Wy_xz
#    0004   16  C.sfr.MICin_DINT_pending  Y   0,0   0:AL:NRESET     ---U   R0_0x R1_11  Wx_0y Wx_11
# -------  ---  ------------------------  -  -----  -----------  --------  -----------  -----------
#    0010    0  C.sfr.PCMin_WADDR[0]      Y   X,X                   ---U   Rx_xy        Wy_xz
#    0010    8  C.sfr.PCMout_WADDR[0]     Y   0,0   0:AL:NRESET     ---U   R0_00 R1_1x  Wx_00 Wx_1y
```

Figure 16. An example log generated by ARBM post-processing program with design C

```
# Address  Bit  Type     Read Write  RC  RS  WC    WS W1C W1S W1T W0C   W0S W0T Wonce RA0 RA1   UBA
# -------  ---  -------  ----------------------  --------------------  ----------------------  ----
# 0000       0  RW          P     P   F   F   F     F   F   P   F   P     F   F     F   F   F     P
# 0000       1  RW          P     P   F   F   F     F   F   P   F   P     F   F     F   F   F     F
# 0000       5  UNDEF1      F     F   F   F   F     F   F   F   F   F     F   F     F   P   F     F
# 0000      24  WO          F     P   F   F   F     F   F   P   F   P     F   F     F   P   F     F
# 0004       0  RO          P     F   F   F   F     F   F   F   F   F     F   F     F   F   F     F
```

Figure 17. An example log generated by the post-processing program for UVM-based back-door exploration with design C