http://dx.doi.org/10.5516/NET.04.2013.080

FUNCTIONAL VERIFICATION OF A SAFETY CLASS CONTROLLER FOR NPPS USING A UVM REGISTER MODEL

KYUCHULL KIM
Dankook University, Dept. of Applied Computer Engineering, 448-701, Korea
E-mail : kckim@dku.edu

Received September 10, 2013
Accepted for Publication December 30, 2013

A highly reliable safety class controller for NPPs (Nuclear Power Plants) is mandatory as even a minor malfunction can lead to disastrous consequences for people, the environment or the facility. In order to enhance the reliability of a safety class digital controller for NPPs, we employed a diversity approach, in which a PLC-type controller and a PLD-type controller are to be operated in parallel. We built and used structured testbenches based on the classes supported by UVM for functional verification of the PLD-type controller designed for NPPs. We incorporated a UVM register model into the testbenches in order to increase the controllability and the observability of the DUT(Device Under Test). With the increased testability, we could easily verify the datapaths between I/O ports and the register sets of the DUT, otherwise we had to perform black box tests for the datapaths, which is very cumbersome and time consuming. We were also able to perform constrained random verification very easily and systematically. From the study, we confirmed the various advantages of using the UVM register model in verification such as scalability, reusability and interoperability, and set some design guidelines for verification of the NPP controllers.

KEYWORDS : Functional Verification, SystemVerilog, UVM Register Model, Safety Class Controller, Diversity, Controllability, Observability

1. INTRODUCTION

Recently conventional analog controllers for NPPs (Nuclear Power Plants) are being replaced by digital controllers due to their maintenance problems[1]. Usually diversity is used in order to avoid CCFs(Common Cause Failures) in the controllers. Using a PLC(Programmable Logic Controller)-type controller based on microprocessors in parallel with a PLD(Programmable Logic Device)-type controller based on CPLD(Complex Programmable Logic Device) or FPGA(Field Programmable Gate Array) is one way of obtaining diversity. To this end, many countries attempted to develop PLD-type controllers to be operated in parallel with PLC-type controllers that were already developed. Ukraine[2], USA[3] and Canada[4] developed I&C(Instrumentation and Control) platforms based on FPGA and adopted it in safety systems such as shutdown systems. We are also developing a PLD-type controller, and trying to verify the controller design in Korea.

We built structured testbenches using the classes based on UVM(Universal Verification Methodology) supported by SystemVerilog in order to verify the design of a PLD-type safety class controller for NPPs and performed a functional coverage analysis. We incorporated a UVM register model into the testbenches. The observability and controllability of the DUT were greatly increased with the register model and we could automatically test the datapaths between I/O ports and the register set of the DUT using a scoreboard and a coverage collector. We were also able to perform constrained random verification very easily and systematically. Another effect of using the register model is that the total verification time can be reduced by partitioning the datapaths. The register model for the DUT(Device Under Test) was reused as a register block in the register model in the integration level test-benches for the parent module. Thus we confirmed its effects in the design verification. With the results of this study, we strongly suggest the use of the register model in the verification of the safety class controllers for NPPs in order to get the reliability complying with IEC 62566. We also suggest the extensive use of register sets in the design, which helps the use of the register model for the design verification by setting the configuration of the design and increasing the testability. To the best of our knowledge, this was the first attempt to incorporate a register model into the testbenches in verifying a controller design for NPPs in Korea.

This paper is organized as follows. In section 2, the structure of the safety class controller, which is the DUT to be verified in this study and its operation explained in some detail. In section 3, we introduce the structures of the UVM testbench and its verification components such as sequencer, driver, monitor, adapter, scoreboard, etc.

Section 4 presents the UVM register model integrated into the testbench for the DUT. In section 5, we explain the effects expected when the register model is used in verifying the DUT design. The verification results are described in section 6, followed by the conclusion in the last section.

2. STRUCTURE AND OPERATIONS OF THE DUT

Fig. 1 shows the simplified basic structure of the designed safety class controller for NPPs. It is composed of FPM, *FDI, FDO, FAI,* and *FAO* modules. The *FPM* is a processor module and all others are I/O modules. In the module names, *F* stands for FPGA, *D* for Digital, *A* for Analog, *I* for Input, and *O* for Output. The controller has one *FPM* and can have up to 16 I/O modules. All I/O modules communicate with the processor module through a serial bus called an SBUS.

All I/O modules have an *SBUS_CTRL* submodule which is essential for communication with the *FPM* through the *SBUS*. Thus we chose it as an example DUT to show how we employed the UVM register model for verification. The *SBUS_CTRL* receives commands from the *FPM*, such as *set, out, check_id, reset, clear, enable, and disable*. *Set* command sends 8 bytes of data fed from the *SBUS* to an output port. *Out* command sends 2 bytes of output signals fed from the *SBUS*. Among the commands *set, out and check_id* send a response signal back to the *FPM* through the *SBUS*. All the other commands are used to control the I/O modules.

The *SBUS_CTRL* has 12 internal registers, *REC_reg00~09*, *CRC_reg*, and *Tx_reg*. Receive registers *REC_reg00~09* hold the data received from *Sin*, an input pin connected to the *SBUS*. The CRC(Cyclic Redundancy Code) register *CRC_reg* holds the CRC code generated for the data stored in the receive registers. Transmit register *Tx_reg* receives data from *REC_reg00~01* or *Stat_Ireg00~05* one by one. The data stored in *Tx_reg* is used to form a message to be sent back to the *FPM* via the *Sout*, an output pin connected to the *SBUS*. This will be explained further in a later section.

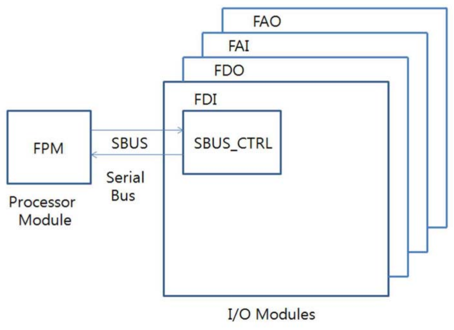


Fig. 1. Diagram of the Designed Controller

3. A SYSTEMVERILOG TESTBENCH USING UVM

We built a SystemVerilog testbench using UVM. Thus we will briefly explain SystemVerilog and UVM in this section. SystemVerilog is an extensive enhancement to the IEEE 1364 Verilog-2001 standard. This enhancement provides powerful capabilities for modeling hardware at the RTL(Register Transfer Level) and system level, along with a rich set of new features for verifying model functionality[6].

UVM stands for Universal Verification Methodology. It combines technologies that originated in Mentor's AVM(Advanced Verification Methodology), Mentor & Cadence's OVM (Open Verification Methodology), Versity's eRM(e Reuse Methodology), and Synopsys's VMM-RAL(Verification Methodology Manual - Register Abstraction Layer), along with new technologies such as Resources, TLM2(Transaction Level Modeling 2) and Phasing. It provides a powerful, flexible technology and methodology to help you create scalable, reusable, and interoperable testbenches[7].

Fig. 2 shows the structure of a UVM testbench. Unlike a conventional testbench, it has various kinds of verification components. The components included in the UVM agent are sequencer, driver, monitor, analysis component, and configuration object. Analysis component and configuration object in agents are not shown for simplicity in the figure. A sequencer denoted by *SQR* in the figure routes sequence-items from a sequence where they are generated to a driver. A **driver** converts the data inside a sequence-item into a pin level transaction. A **monitor** observes pin level activity and converts its observations into sequence-items which are sent to components, such as scoreboards, which in turn use them to analyze what is happening in the DUT.

In a block level UVM testbench, the environment (denoted by ENV) contains the agents needed to communicate with the DUT's interfaces in one place, as shown in Fig. 2. The ENV may also contain a **configuration object**, scoreboard, coverage monitor, and virtual sequencer. The

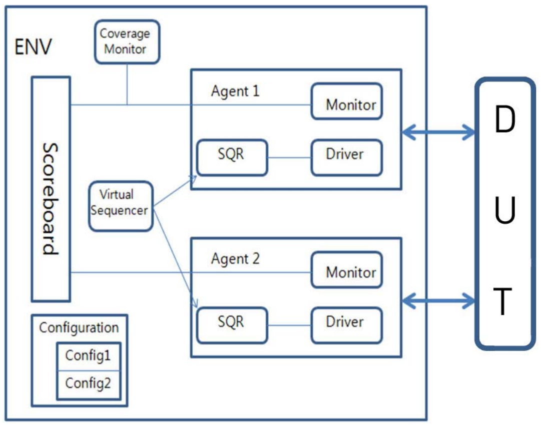


Fig. 2. Structure of a UVM Testbench

configuration object enables the test writer to control which of the ENV's sub-components are built. It also contains a handle to the configuration object for each agent that it contains. A **scoreboard** is an analysis component that checks if the DUT is behaving correctly. It uses analysis transactions received from the monitor. It usually compares transactions from at least two agents. A **coverage** monitor contains covergroups to gather functional coverage information. A **virtual sequencer** is used in the stimulus generation process to allow a single sequence to control activity via several agents.

You can refer to the UVM cookbook for further information about UVM testbenches explained in this section[7].

4. UVM REGISTER MODEL FOR THE *SBUS_CTRL*

The UVM register model provides a way of tracking the register content of a DUT and a convenience layer for accessing register within the DUT. It reflects the structure of a hardware-software register specification. It is designed to facilitate productive verification of programmable hardware. Thus, the level of stimulus abstraction can be increased and the resultant stimulus code becomes easy to reuse, either when there is a change in the DUT register address map, or when the DUT block is reused as a sub-component[6].

Fig. 3 shows the structure of a UVM testbench with a register model which adopts an explicit prediction method[7]. *SQR* stands for sequencer, *DRV* for driver, and *MON* for monitor in the figure. The monitor observes a bus transaction and sends the corresponding bus sequence item to the predictor. The predictor looks up the accessed register, and then updates the register contents of the register model. Note that the actual testbench may contain multiple bus agents, one for each interface.

The register model abstraction should reflect the structure of a hardware-software register specification. Thus, the details of the registers contained in the *SBUS_CTRL*,

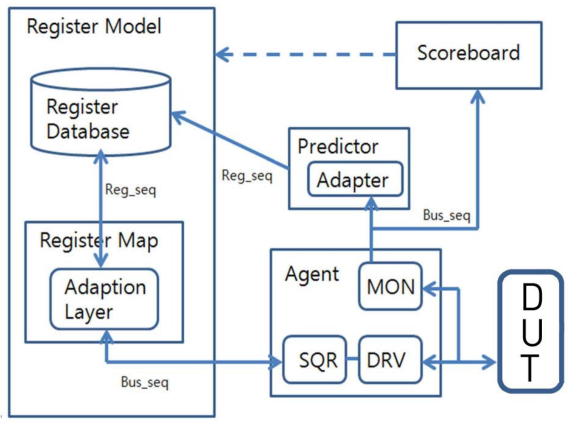


Fig. 3. UVM Testbench with Register Model

which is an example DUT, will be explained in the following section.

Fig. 4 shows I/O signals and registers of the *SBUS_CTRL*. According to the characteristics and the protocols of the I/O pins, we categorized the pins into 7 interfaces and built 7 agents for each interface respectively. Table 1 shows the name of the agents and their associated I/O signals.

The *SBUS_CTRL* has 12 registers which are denoted by *REC_reg00~09*, *CRC_reg*, and *Tx_reg*. Receive registers *REC_reg00~09* receive data from the *SBUS* through *Sin*. A message coming from the *FPM* has 4 fields, *FUNCTION*, *ID*, *DATA*, and *CRC*. *FUNCTION*, *ID*, and *CRC* fields are 16 bits long. The *FUNCTION* field is composed of 3 subfields, *command*, *sel_bus*, and *pos_slot*, which occupy 8 bits, 2 bits and 4 bits respectively. The remaining 2 bits are padded with 0's. The *DATA* field is n * 16 bits long, where n = 2 for the *out* command, n = 8 for the *set* command, and n = 0 for all other commands. Fig. 5 shows the fields of incoming/outgoing messages.

Upon receiving a message from the *FPM*, the DUT stores the *FUNCTION* field in *REC_reg00*, the *ID* field in *REC_reg01*, and the *DATA* field in *REC_reg02~09*.

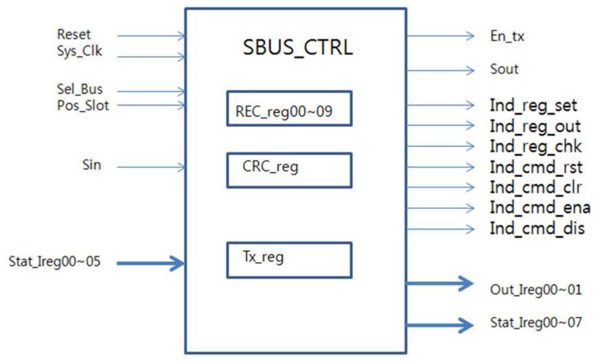


Fig. 4. I/O Signals and Registers of the SBUS_CTRL

Table 1. Agents and Associated I/O Signals of the SBUS_CTRL

Agent	I/O signals
Basic	Reset, Sys_Clk
Config	Sel_Bus, Pos_Slot
Sbus	Sin, Sout
Status	Sta_Ireg00~05
Indicator	Commands
Out	Out_Ireg00~01
Set	Set_Ireg00~07

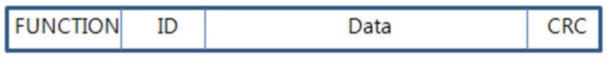


Fig. 5. Fields of Incoming/Outgoing Messages

The number of registers used to store the *DATA* field varies according to its size. The *CRC_reg* stores the CRC code generated from the message received. The *SBUS_CTRL* also has a *Tx_reg*, which receives contents of receive registers *REC_reg00~01* or status registers *Stat_Ireg00~05*, one at a time.

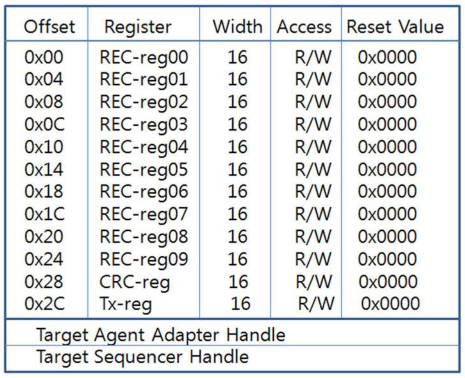
After receiving a message from the *FPM*, the *SBUS_CTRL* sends a response message back to the *FPM* through the *SBUS* for the three commands of *check_id*, *set*, and *out*. The *Tx_reg* is used to form this response message. In the response message the *DATA* field is filled with the information from the status registers *Stat_Ireg00~05*. A CRC code is generated from the contents of the *DATA* field in the message and is appended to the message.

The *SBUS_CTRL* sends the contents of the *REC_reg02* and *REC_reg03* to the *Out_Ireg00* and *Out_Ireg01* output ports respectively for the *out* command and the *REC_reg02~09* to the *Set_Ireg00~07* output ports respectively for the *set* command.

Fig. 6 shows the contents of the register map built from the above description. It has *offset*, *width*, *access mode*, and *reset value* for each registers and handles to the target agent adapter and target sequencer. In this model, all registers have the same width, access mode, and reset values.

The register model provides an API (Application Programming Interface) for sequences to access software mapped hardware registers. It also mirrors the content of the hardware. Refer to Fig. 3 again. The adapter converts register transactions (*Reg_seq*) to bus sequence items (*Bus_seq*) and vice versa. The predictor receives all the bus sequence items observed by the monitor attached to the interface and converts them to register transactions and updates the register database with them.

Scoreboard is an analysis component. It has a handle to the register model in order to access the register values. Scoreboard compares observed data against register contents and/or DUT contents against expected data via backdoor access *peek*. By doing these comparisons, scoreboard determines the correctness of the operations of the DUT.



Offset	Register	Width	Access	Reset Value
0x00	REC-reg00	16	R/W	0x0000
0x04	REC-reg01	16	R/W	0x0000
0x08	REC-reg02	16	R/W	0x0000
0x0C	REC-reg03	16	R/W	0x0000
0x10	REC-reg04	16	R/W	0x0000
0x14	REC-reg05	16	R/W	0x0000
0x18	REC-reg06	16	R/W	0x0000
0x1C	REC-reg07	16	R/W	0x0000
0x20	REC-reg08	16	R/W	0x0000
0x24	REC-reg09	16	R/W	0x0000
0x28	CRC-reg	16	R/W	0x0000
0x2C	Tx-reg	16	R/W	0x0000

Target Agent Adapter Handle
Target Sequencer Handle

Fig. 6. Contents of the Register Map

5. EFFECTS OF USING A REGISTER MODEL IN DUT VERIFICATION

We assumed the following effects from applying the register model in the design verification of the NPP controller.

1) Improvement in testability

The register values can be checked or set only through indirect methods if the register model is not applied. However, the register model allows for the confirmation and setting of the register values directly. In other words, the testability increase by applying the register model is similar to improving testability by inserting the test points in the board test.

a) Observability

It is particularly more difficult to read the register value when the register is a write-only register such as the configuration register of a programmable module. The register value can be read directly without consuming simulation time by using a backdoor *peek* command in the testbench when the register model is used. Thus the DUT's observability is improved.

b) Controllability

The status register of a DUT is a read-only register. In this case, the controllability is low as the status register value is determined by the DUT state and not set directly by the test engineer. However, controllability can be improved by employing the register model as the status register value can be changed directly using the backdoor *poke* command in the test bench.

2) Easy Diagnosis

When the verification results indicate there is an error in the DUT, it needs to be clarified which part in specific caused the malfunction. As shown in Fig. 7, the datapaths between the input port and the register and the datapaths between the output port and the register are tested in the register test when the register model is applied. Therefore, the malfunctioning datapath can be detected immediately when an error occurs.

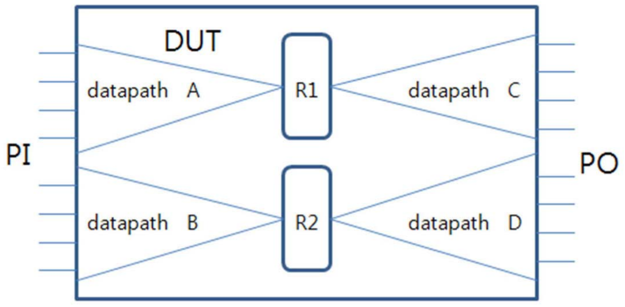


Fig. 7. Datapath Diagnosis

3) Verification Time Reduction
From Fig. 7, we can see that the datapaths are segmented into smaller ones. Thus, applying the register model in the verification process can be thought to be equivalent to employing the 'divide and conquer' strategy. Therefore, if the verification time is longer than *0(n)*, the time required in the verification of the entire DUT decreases when the register model is used. For example, assume the time required in the verification of the entire DUT is n^2. If the circuit is divided into 4 subsections, and the verification is made separately for each subsection, the time required for the verification is $4 * (n/4)^2 = n^2/4$, thus the verification time decreases by 1/4.

4) Easy Generation of Cover Group
The cover group and cover point need be generated appropriately in order to obtain meaningful functional coverage. If such is generated based on the register model, a more meaningful functional coverage can be calculated. Therefore, the efforts required in verification may be reduced.

6. VERIFICATION RESULTS USING THE UVM REGISTER MODEL

In order to utilize the benefits explained in the previous section, we built a UVM testbench with a structure similar to the one shown in Fig. 3. We incorporated the register model explained in section 4. The testbench has 7 agents for 7 interfaces respectively. We performed verification of the DUT using Mentor Graphic's QuestaSim 10.2a from which they began supporting HDL paths for the designs written in VHDL. So we could use backdoor access to the DUT written in VHDL.

Constrained randomization is a great way to find bugs quickly and explore a design's state space thoroughly. The best way to check the correctness of the DUT's behavior is to use a monitor on each interface, which sends observed transactions to a scoreboard where the behavior is then checked for correctness. This frees the stimulus to be more flexible, reusable, and random[8]. SystemVerilog also supports automatic generation of random test cases using a random test method[5].

We randomized the input stimuli for constrained random verification as follows. Inside the *FUNCTION* field, there are 3 subfields: *command*, *sel_bus*, and *pos_slot*. The *command* can be one of *check_id*, *set*, *out*, *reset*, *clear*, *enable*, and *disable*. The *sel_bus* and *pos_slot* have full random values. *ID* is also fixed to a constant specific to the I/O module, for example, '44F1' for the *FDO* module. The *DATA* field and parallel input *Stat_Ireg00~05* are fully randomized without constraint.

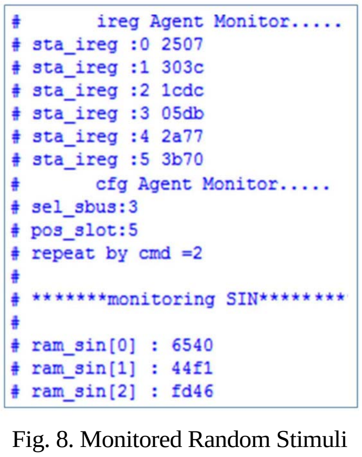
The Monitor sends the observed transactions on input pins to the predictor in order to update the register values in the register model. It also sends the observed transactions to the scoreboard.

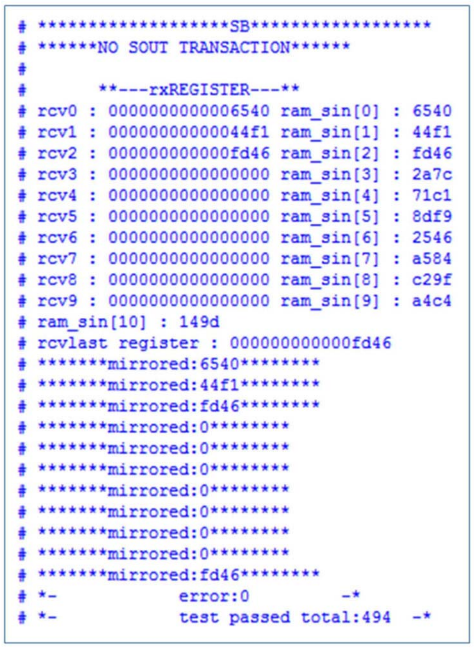
The scoreboard peeks the *SBUS_CTRL* for hardware register values and compares them with the sequence items sent from the monitor. Upon receiving sequence items for the output pins from the monitor, the scoreboard compares them with the mirrored values stored in the register model. It also checks the command indicator output signals by identifying the command stored in the *REC_reg00*. In this way the scoreboard is able to determine the correctness of the DUT's operation. This process is repeated until predetermined satisfactory coverage is attained, which tells us that the test is complete.

Fig. 8 shows the stimulus values observed by three agent monitors: *Status* agent, *Config* agent, and *Sbus* agent. These three agents send input stimuli to the *SBUS_CTRL* at the same time.

Fig. 9 shows a snapshot of the scoreboarding process. The upper part shows the peeked values of the DUT registers and the lower part shows the mirrored values stored in the database of the register model. Peek is a backdoor access method which directly accesses the simulator's database to get the DUT's register values without using simulation time. Note that only 4 registers are updated by this transaction in the figure. The 4 registers are *REC_reg00~02*(denoted by *rcv0~2* in the figure) and *CRC* register (denoted by *rcvlast* register in the figure). Scoreboard compares these values to determine the correct operation of the *SBUS_CTRL*. Note that all 4 mirrored values match the 4 peeked values respectively. Also note that the rest of the registers *REC_reg03~09*(denoted by *rec3~9* in the figure) hold the old values from the previous transaction. By doing this type of test we could verify the integrity of the data paths from input ports to the register set. We also performed the test to verify the integrity of the data paths from the register set to output ports.

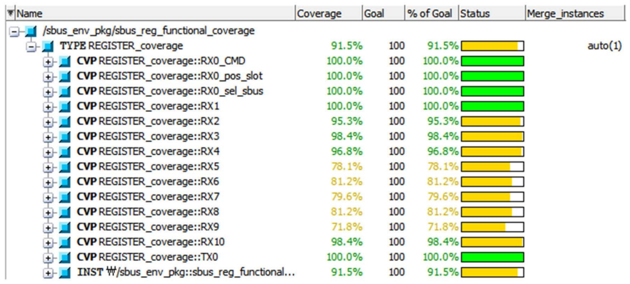
Fig. 10 shows the coverage report for the register model. In the figure, *RX0~9* corresponds to *REC_reg00~09* respec-


```
#       ireg Agent Monitor.....
# sta_ireg :0 2507
# sta_ireg :1 303c
# sta_ireg :2 1cdc
# sta_ireg :3 05db
# sta_ireg :4 2a77
# sta_ireg :5 3b70
#       cfg Agent Monitor.....
# sel_sbus:3
# pos_slot:5
# repeat by cmd =2
#
# *******monitoring SIN********
#
# ram_sin[0] : 6540
# ram_sin[1] : 44f1
# ram_sin[2] : fd46
```
Fig. 8. Monitored Random Stimuli



```
# ******************SB*****************
# ******NO SOUT TRANSACTION******
#
#       **---rxREGISTER---**
# rcv0 : 0000000000006540 ram_sin[0] : 6540
# rcv1 : 00000000000044f1 ram_sin[1] : 44f1
# rcv2 : 000000000000fd46 ram_sin[2] : fd46
# rcv3 : 0000000000000000 ram_sin[3] : 2a7c
# rcv4 : 0000000000000000 ram_sin[4] : 71c1
# rcv5 : 0000000000000000 ram_sin[5] : 8df9
# rcv6 : 0000000000000000 ram_sin[6] : 2546
# rcv7 : 0000000000000000 ram_sin[7] : a584
# rcv8 : 0000000000000000 ram_sin[8] : c29f
# rcv9 : 0000000000000000 ram_sin[9] : a4c4
# ram_sin[10] : 149d
# rcvlast register : 000000000000fd46
# *******mirrored:6540********
# *******mirrored:44f1********
# *******mirrored:fd46********
# *******mirrored:0********
# *******mirrored:0********
# *******mirrored:0********
# *******mirrored:0********
# *******mirrored:0********
# *******mirrored:0********
# *******mirrored:0********
# *******mirrored:fd46********
# *-          error:0         -*
# *-          test passed total:494   -*
```

Fig. 9. A snapshot of the Scoreboarding Process



Name	Coverage	Goal	% of Goal	Status	Merge_instances
/sbus_env_pkg/sbus_reg_functional_coverage					
TYPE REGISTER_coverage	91.5%	100	91.5%	🟨	auto(1)
CVP REGISTER_coverage::RX0_CMD	100.0%	100	100.0%	🟩	
CVP REGISTER_coverage::RX0_pos_slot	100.0%	100	100.0%	🟩	
CVP REGISTER_coverage::RX0_sel_sbus	100.0%	100	100.0%	🟩	
CVP REGISTER_coverage::RX1	100.0%	100	100.0%	🟩	
CVP REGISTER_coverage::RX2	95.3%	100	95.3%	🟨	
CVP REGISTER_coverage::RX3	98.4%	100	98.4%	🟨	
CVP REGISTER_coverage::RX4	96.8%	100	96.8%	🟨	
CVP REGISTER_coverage::RX5	78.1%	100	78.1%	🟨	
CVP REGISTER_coverage::RX6	81.2%	100	81.2%	🟨	
CVP REGISTER_coverage::RX7	79.6%	100	79.6%	🟨	
CVP REGISTER_coverage::RX8	81.2%	100	81.2%	🟨	
CVP REGISTER_coverage::RX9	71.8%	100	71.8%	🟨	
CVP REGISTER_coverage::RX10	98.4%	100	98.4%	🟨	
CVP REGISTER_coverage::TX0	100.0%	100	100.0%	🟩	
INST ₩/sbus_env_pkg::sbus_reg_functional...	91.5%	100	91.5%	🟨	

Fig. 10. Coverage Report for the Register Model

tively. *RX-10* is *CRC_reg* and *TX0* is *Tx_reg*. For each message the *SBUS_CTRL* receives a command which is selected randomly among 7 commands. All commands use *REC_reg00~01* and *CRC_reg* in common. In addition to *REC_reg00~01* and *CRC_reg*, *check_id* uses *REC_reg02*, out uses *REC_re02~04*, and *set* uses *REC_reg02~09*. That is why *REC_reg00~01* have highest coverage, *REC_reg02~04* have higher coverage than *REC_reg05~09*. Overall register coverage obtained is 91.5%. Note that for some registers such as *REC_reg00~01*, which contain *command*, *sel_bus*, *pos_slot*, and *module_id*, 100% coverage is necessary. However, the registers such as *REC_reg02~09*, which contain data, do not require 100% coverage. If the coverage for the registers *REC_reg02~09* is considered too low, we can do additional guided random testing to increase the coverage by modifying the randomization constraints. This is a very powerful feature of the constrained random testing.

7. CONCLUSION

We implemented a UVM testbench to verify the design of a safety class controller for nuclear power plants. We incorporated a UVM register model into the testbench and performed the constrained random testing for verification. Constrained random testing is a very useful way to find bugs quickly and thoroughly explore a design's state space. We also collected functional coverage for the register model using a functional coverage monitor. With the coverage collector, we can perform adaptive random testing to increase the register coverage to a satisfactory level. From the study we confirmed that backdoor peek access supported by a UVM register model is very useful for functional verification of a DUT with registers, especially when the register is volatile. We also confirmed that the register model is scalable, reusable, and interoperable in integration level testing where the DUT is composed of multiple modules including *SBUS_CTRL*. In this case, the upper level register model contains the register model for *SBUS_CTRL* as a register block. We can conclude that the UVM testbench with a register model is a very strong and powerful tool for functional verification of safety class controllers for nuclear power plants, which require a high level of reliability.

ACKNOWLEDGEMENT

The present research was conducted by the research fund of Dankook University in 2012.

REFERENCES

[1] Patrick Salaün, Frederic Daumas, Thuy Nguyen, and Claude Esmenjaud, “FPGA/ASIC: A promising technology for future of I&C Systems in power industry,” 6th American Nuclear Society International Topical Meeting on NPIC&HMIT, April, Knoxville, Tennessee, USA, 2009.
[2] Vyacheslav Kharchenko, “Experience of RPC <<Radiy>> is designing, manufacturing and implementation of FPGA-based NPP I&C systems,” 1st Workshop on The Applications of Field-Programmable Gate Arrays in Nuclear Power Plants, October, Chatou, France, 2008.
[3] Bernard F. Dittman, “Regulatory Experience with a FPGA-based Digital I&C Review,” 2nd Workshop on The Applications of Field-Programmable Gate Arrays in Nuclear Power Plants, September, Kirovagrad, Ukraine, 2009.
[4] Jingke She and Jin Jiang, “Application of FPGA to Shutdown system No. 1 in CANDO”, 6th American Nuclear Society International Topical Meeting on NPIC&HMIT, April, Knoxville, Tennessee, USA, 2009.
[5] Chris Spear, “SystemVerilog for Verification: A Guide to Learning the Testbench Language Features,” 2nd Edition, Springer, 2008.
[6] S. Surtherland, An Overview of SystemVerilog 3.1, Web : http://www.eetimes.com/ document.asp?doc_id=1277143.
[7] Verification Methodology Cookbooks Web: https:// verificationacademy.com/cookbook/ Cookbook.
[8] Getting Started with OVM, Web: http://www.doulos.com /knowhow/sysverilog.