Test environment for verification of multi-processor interrupt system with virtualization support

Dmitriy Lebedev
Department of Verification and Modeling
MCST
Moscow, Russia
lebedev_d@mcst.ru

Vitaliy Kutsevol
Department of Verification and Modeling
MCST
Moscow, Russia
kutsevol_v@mcst.ru

***Abstract* —Modern microprocessor interrupt systems include hardware support for virtualization. Hardware support helps to increase the performance of virtual machines. However, including additional functionality may lead to potential errors. The paper presents an overview of approaches used for multi-core microprocessors interrupt system verification. Some definitions and characteristics of interrupt systems that needed to be taken into account in the process of verification are described. Stand-alone verification environment general scheme is presented. Universal Verification Methodology was applied to construct test system. We describe some difficulties discovered in the verification process and corresponding solving methods. Generalized test algorithm stages are presented. Some other techniques for checking the correctness of interrupt system have been reviewed. In conclusion, we provide the case study of applying the suggested approaches for interrupt system verification of microprocessors with "Elbrus" and "SPARC-V9" architectures developed by MCST. The results and further plan of the test system development are presented.**

Keywords — test environment, standalone verification, multicore microprocessors, interrupt system, UVM, virtualization.

I. INTRODUCTION

State of the art microprocessors are becoming complex systems. The development of new technological processes allows integrating great number of controllers and subsystems on the one crystal. The logic of their work becomes more complicated. As an example: interrupts delivery and handling mechanisms. Interrupts are widely used for interaction with hardware and responding to stimuli. Interrupt system is an important part of microprocessor and have to be tested thoroughly because errors in this block may lead to a race condition or multiple accesses to shared memory [1].

Virtualization is necessary for modern tasks like cloud computing, modeling, information security, and other scientific researches. Interrupt system hardware support for virtualization is implemented in the most of modern SoC (System on a Chip) [2-3]. Performance requirements for virtualized systems are steadily increasing [4]. One method to increase performance of virtual operating system is to implement hardware support. However, this becomes an additional error-prone place in the microprocessor system. In addition, I/O virtualization is a difficult part of system virtualization due to the increasing number of I/O devices attached to the computer and the increasing diversity of I/O device types [5]. A significant part of the total number of interrupts is accounted for I/O interrupts.

Many approaches are proposed for verification of the interrupt system. In [6] authors divide existing techniques for verifying interrupt systems into two categories. First of them is testing via executing some programs and invocation various interrupts. The disadvantage of this approach is probability of missing important bugs. The second one is constructing and analyzing formal models. As was already mentioned in [7] constructing and analyzing of formal models are complicated and requires detailed specification. In this paper, we propose simulation-based methods for verification of interrupt system that are much simpler and could be applied at earlier stage of RTL (Register Transfer Level) model development when first versions of specification were available.

In [8] verification of the interrupt system provided using interrupt-driven software which launched on the whole microprocessor system. This method gives good results in finding race condition bugs but requires fine-tuning of interrupt sending schedule. Moreover, none of the discussed methods recreates rare dynamic scenarios. Stand-alone verification usually used for building necessary test conditions to achieve sufficient testing quality.

There are a number of methods to implement a standalone functional verification [7]. In this paper, we focus on building testing environment for interrupt system using Universal Verification Methodology (UVM) [9]. UVM is IEEE standard elaborated by Accellera Systems. UVM is a set of class libraries defined using the syntax and semantics of SystemVerilog hardware description and verification language. The verification environment built using UVM divided into specific components each of them performs its own role in test scenario. One of the main advantages of UVM-build verification environments is reusability of components. It helps to support probable changes in DUV (Device Under Verification). The main drawback of UVM is a complexity of its learning. Our verification team have a number of already debugged components and classes. Therefore, we can apply UVM for developing interrupt system stand-alone verification environment.

The rest of the paper is organized as follows. Section 2 reviews some definitions and describes general technique for standalone verification of the interrupt subsystem. Section 3

describes a case study and suggests approaches for functional verification of interrupt system. Section 4 describes additional used approaches. Section 5 reveals results and Section 6 concludes the paper.

II. DEFINITIONS AND VERIFICATION METHODS

Let us give a few definitions. An i*nterrupt* is asynchronous signal that indicates necessity for control transferring to some external to the processor core requester. An *interrupt vector* characterizes the transmitted signal. Interrupt vector points to the memory area where the corresponding *interrupt handler* is located. The interrupt handler is a code that should be executed instead of current main program. It essential to mention that interrupts change main memory and device registers state but main context of processor work stays the same.

We can divide interrupts into two types: *non-maskable* and *maskable*. Non-maskable interrupt cannot be disabled. It has more priority then maskable and used for exceptional conditions like critical faults, system handling and other. Maskable interrupt is intended for maintenance of system and user programs: the organization of external exchanges, interaction of different processes and working with timers. Maskable interrupts usually have several levels of priority. If an interrupt signal is received, but its priority is less than the one that came earlier, the interrupt becomes *pending*. Important time characteristic is *interrupt latency*. This is a time interval from the start of the interrupt request to the start of the interrupt handler execution. A set of system setups such as interrupt enabling, current priority, and the presence of a large number of pending interrupts can cause an *interrupt loss* or *spurious interrupt*. The spurious interrupt is an invalid, short-duration signal on an interrupt input. It is necessary to take into account these features when verifying interrupt system.

It is necessary to define some concepts related to virtualization. *Hypervisor* is a system software that distributes physical resources between *virtual machines*. A computer on which a hypervisor handles one or more virtual machines is called a *host*, and each virtual machine is called a *guest*. As a part of hardware interrupts support, the guest can be bounded on one or more cores. These cores are called *guest cores*. Without hardware support, the virtual software model of the interrupt controller, register requests, and interrupt delivery involved interception and emulation. Because of this, the performance of the VM is reduced.

We isolate a part of microprocessor system while providing stand-alone verification. The device specification have to describe correct sequence of stimuli and reactions in different device states. All interactions controlled by test environment - a program that generates input stimuli, checks correctness of reactions and calculates the quality of testing. Therefore, test environment could be divided into separate modules each performing its own function:

- input stimuli generator;
- correctness checking module;
- coverage collector.

Usually the interrupt controllers are handled by a set of software-visible registers. All requests to registers processed sequentially strictly one at a time. Thus, input generator is responsible not only for sending primary requests it also sets up a device. Next, we need to collect device responses and process them correctly. Generation of stimulus and collection of reactions simplified by using Transaction Level Modeling (TLM) [10]. This method allows concentrating on the interaction functionality with DUV. It is necessary to implement only once how to handle with interface and then simple data sending and receiving functions are used. Collected information about functional code coverage is used to identify untapped regions of controller during testing. Analyzing that information helps to refine test scenarios and add new ones. This approach is called coverage driven constrained random verification [11].

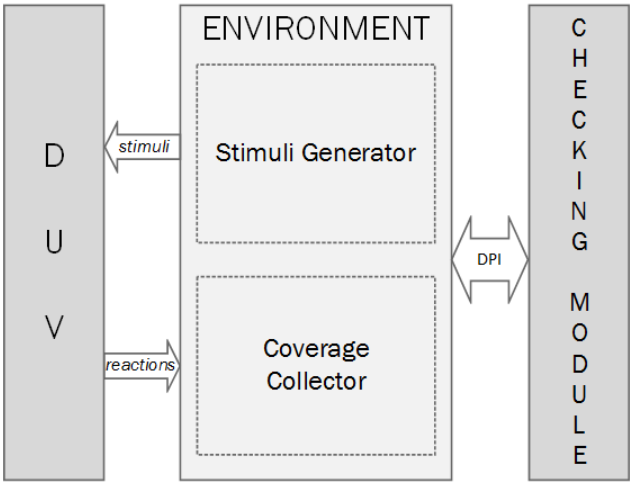

Fig 1. Generalized scheme of test environment

Checking of behavior correctness is a complex and an important part while providing interrupt system stand-alone verification. It is usually easier to build test environment external to the reference model for this purpose. Identical input stimuli fed into the DUV and reference model. If reactions differ, it indicates a possible error in the system. Reference models usually written in high-level language (C, C++) or some specialized languages for hardware verification, such as SystemVerilog, SystemC or «e». The reference models could be divided into three types: *cycle-accurate, discrete-event with time accounting* and *event models* [12]. The type of verified device defines the type of the reference model. Cycle-accurate and discrete-event with time accounting models require specification with describing behavior on a register transfer level. Development of models of these types is labor-intensive when the design specification is changing. However, because the interrupt system includes the use of counters, we have to choose more accurate reference model. To simplify development of checking module we use discrete-event with time accounting reference model. Communication between reference model and test environment carried out with DPI (Directed Programming Interface). The use of DPI is necessary to coordinate the variable types used in different programming languages. Exchange of test data occurs instantly by calling

appropriate functions. Generalized scheme of test environment shown on fig. 1.

III. FUNCTIONAL VERIFICATION OF INTERRUPT SYSTEM

Elbrus Programmable Interrupt Controller (EPIC) is a device intended for capturing, storing external and interprocessor interrupts and delivering them to the microprocessor cores. EPIC is a part of 16-core microprocessor with "Elbrus" architecture developing by MCST. Software-available registers manage the controller setup and handling of interrupts. The algorithms for working with maskable and non-maskable interrupts are different. Maskable interrupts have four priority classes. An unhandled interrupt with the highest priority are placed on CIR (Current Interrupt Register) or if CIR is busy placed on PMIRR (Pending Maskable Interrupt Request Registers). Signal to the core sets only if current core priority is less than interrupt priority in CIR register. Interrupt system implements hardware support of virtualization. Supporting options include additional control registers representing the state of guest interrupt system and guest-physical core mapping table.

As we mentioned before, test system for stand-alone verification of the interrupt system is based on UVM. UVM helps to setup system, generate pseudo-random constrained input requests and monitor all changes of device states. Input stimuli generation is usually implemented at level that is more abstract than register transfers and interface signals. Input and output device ports combined into interfaces based on the similarity of the performed functions. Transaction level packets are transferred on interfaces [13]. Serialization and deserialization modules transform transaction level packets to signal level interfaces.

In modern microprocessor system, there are several sources of interrupt requests or primary requests. Each interrupt source is replaced by a special test sequence. Generation of primary requests should be similar to that in real microprocessor system. One of the benefits of stand-alone verification is ability to create highly loaded test scenarios relatively easily. The efficiency of verification increases because generated pseudo-random requests can cover most of the edge cases. We use sequences of primary requests and generate secondary requests automatically in the special modules named *auto-handlers*. In case of interrupts system secondary request is a writes and reads sequence to the registers that simulating a real software behavior. Interrupts generated from primary requests are monitored and collected in special buffers. After that auto-handler randomly choose next interrupt to handle. Thus, an interrupt latency parameter randomly changes.

For virtualization support, the hardware implements a DAT (Destination Address Table) that stores the number of the currently active guest core for each physical core. Hypervisor manages the table by requests to the registers. The state of the table has to be the same in all processors of multiprocessor system. This is implemented through hardware-generated control messages. Auto-handler module with randomized parameters of sending service messages were added to verify the described above functionality.

Maskable and non-maskable interrupt handling is differ. For maskable interrupts, it is necessary to restore the previous core priority after handling current interrupt. For this purpose, we add monitoring module with memory where ratio "number of core-current priority" stored.

There was a dynamic inconsistency problem between the RTL implementation of the controller behavior and the reference model. Under dynamic test conditions, there may be situations when we start to handle a maskable interrupt *m.handle(x)*, read vector value from RTL-model register and it is not equal to reference model value. For this situation, we introduce additional function *m.correct(x)* that transfer RTL-model value to the reference model. During a test, the reference model accumulates possible vectors. When there is a values inconsistency situation the special algorithm in reference model iterates over possible vector values and makes a decision about correction. The pseudo-code of the algorithm is presented below.

Vector correctness check:

```
while true do
    wait m ← start(x)
    m.handle(x)
    if check(x !=x`) then
      m.correct(x)
    else m.print_ok(x)
end
```

Register for non-maskable interrupts contains several types of interrupts and they represented as one bit for an interrupt. In high-load dynamic tests with many non-maskable input requests for interrupts there may be differences in a register content. A similar procedure for correction *nm.correct(x)* is performed for non-maskable interrupts. From a functional point of view, handling guest interrupts does not differ from the procedures described above. The only difference is setting the bit that indicates a guest when working with registers.

Another problem is an interrupt vectors overlay. This is a subtype of dynamic inconsistency problem when the same vector values are imposed on each other. This can happen for example when working with cyclic timers. Additional functionality was added to the RTL and reference models of verified device. A special module and interface signaling about interrupt overlay monitored in the test environment and then transferred to the reference model. Method when we use hints from a verified device for single correct state identification named "gray box" method [7, 11]. The information about overlay type used by reference model to exclude extra interrupts. This method required direct involvement of the device developer and a detailed description of the interface.

In a multiprocessor system, the interrupt system settings in each processor may differ. It is possible to configure the presence and numbers of processors and cores. It is necessary to check all possible system setups for completeness of verification. Interrupt routing changes when we change the processor or core numbering. This adds restrictions on generating primary requests. To simplify primary requests generation we added a set of functions that automatically

capture changes in system settings and allow easily select possible interrupt directions.

The generalized test algorithm is presented below:

1. randomization of device configuration;
2. configuring registers;
3. configuring guest cores;
4. choosing random requester and presence of receiver;
5. choosing random type of interrupt;
6. sending primary requests for interrupt, collecting reactions, starting auto handling;
7. transferring transaction information to reference model on each step of algorithm;

IV. Additional verification methods

A. Special cases

To check the interrupt system functionality related to virtualization support it was necessary to create special tests scenarios. For example, hypervisor can remove a guest from core if it receives an intercept signal. In this moment guest may contain unfinished or unhandled interrupts. In real system after bounding the guest back, we needed to recover it previous state. Special test cases of this kind play a big role in verifying correctness of controller operating. The development of algorithms for such narrowly focused tests is possible due to the availability of well-described documentation, analysis of the functional coverage of the code, and discussions of the strategy with the device developer.

B. Assertions

SystemVerilog Assertions (SVA) is a part of SystemVerilog [14]. The assertions are used to specify the behavior of the test environment and DUV interfaces. Parts of a verification environment have to implement certain functions. We can add assertions to check correctness of the module. Violation of an assertion signals about an error. Usage of assertions is an effective method of error detection especially in the beginning of the project. In addition, assertions alert about uncertain and unconnected states of interface signals.

C. After test checking

Communication between test environment and reference model is provided using DPI. Special buffers and memories contain generated answers form different interfaces. The correct behavior of the DUV and reference model determined in providing certain number of responses. After test scenario ending we check an absence of transactions in these communication buffers. Detection of extra number of requests signals about a potential error either in the verified device or in the reference model.

V. Results

The approaches described in this paper were applied for standalone verification of interrupt system of the 16-core microprocessor with “Elbrus” architecture and 2-core microprocessor with “SPARC-V9” architecture.

There are some difference in interrupt systems in these microprocessors. The 16-core microprocessor’s interrupt system has a hardware support of virtualization and specialized interfaces for handling requests from virtual OS and hypervisor.

The 2-core microprocessor does not support virtualization. Most of interfaces differs from the “Elbrus” interrupt system. One of its features is an additional module that handles direct MSI-X interrupts and a separate register interface for handling MSI-X interrupts. The test environment based on UVM made it relatively easy to change the format of modules that work with the interfaces while preserving their functionality.

“SPARC-V9” implementation of the interrupt system contains an additional synchro signal. The parts of the interrupt system in which several synchro signals interact should be checked carefully. At the beginning of each test, we generate random periods of synchro signals and their shifts that are relative to each other. Ranges of each synchro signal have to be described in the device specification. This method helped us to detect synchronization errors in RTL-model internal modules.

In the process of the standalone verification of the interrupt systems, we verified not only RTL-models. Parts of reference models were used in full-system “Elbrus” and “SPARC” machine simulators. Aforementioned approaches helped to find and correct some errors in the simulators. Interrupt systems distribution of errors is presented in Table 1. Code functional coverage was carried out and for “Elbrus” it was 94%, for SPARC 96% coverage was extracted.

TABLE 1. DISTRIBUTION OF ERRORS AND ITS QUANTITY

Verified object	Number of bugs
RTL Elbrus	84
Elbrus Simulator	163
RTL SPARC	24
SPARC Simulator	23

VI. Conclusion and directions for future work

Interrupt system with hardware virtualization support is one of important parts of modern microprocessors. The correct operation of the interrupt system allows avoiding critical errors and improves performance of virtual machines.

In this paper, we have presented a stand-alone test environment for interrupt system based on UVM. The proposed approaches could be used to verify interrupt systems of different multicore microprocessors regardless of their architectures. Developed test environment and test scenarios made it possible to detect and correct a number of errors that were not detected by other verification methods.

In the future, we plan to enhance error diagnostics and adapt the test environment for the forthcoming projects.

References

[1] Makoto Higashi, Tetsuo Yamamoto, Yasuhiro Hayase, Takashi Ishio, and Katsuro Inoue. An effective method to control interrupt handler for data race detection. In Proceedings of the 5th Workshop on Automation of Software Test, pages 79–86, 2010.

[2] ARM® Generic Interrupt Controller Architecture Specification version 4.0,2107 https://static.docs.arm.com/ihi0069/c/IHI0069C_gic_architecture_specification.pdf (12.04.2020).

[3] Intel Virtualization Technology for Directed I/O, Architecture Specification. Intel, 2019 https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf (12.04.2020).

[4] Znamensky D.V. Alternatives of hardware virtualization support implementation for elbrus processor architecture. http://www.mcst.ru/files/5345a0/320cd8/501670/000000/znamenskiy-vybor_variantov_realizatsii.pdf (12.04.2020).

[5] Hennessy J.L., Patterson D.A. Computer Architecture: A Quantitative Approach. Fifth Edition. Morgan Kaufmann, 2012. 857 p.

[6] Sung, Chungha & Kusano, Markus & Wang, Chao. (2017). Modular Verification of Interrupt-Driven Software. arXiv:1709.10078v1 [cs.PL] 28 Sep 2017.

[7] Lebedev D.A., Petrochenkov M.V. Test environment for verification of multi-processor memory subsystem unit. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 67-76. DOI: 10.15514/ISPRAS-2019-31(3)-6

[8] John Regehr. Random testing of interrupt-driven software. In International Conference on Embedded Software, pages 290–298, 2005.

[9] Standard Universal Verification Methodology http://accellera.org/downloads/standards/uvm (12.04.2020).

[10] Kamkin A., Chupilko M. A TLM-based approach to functional verification of hardware components at different abstraction levels. Proc. of the 12th Latin-American Test Workshop (LATW), 2011, pp. 1-6.

[11] Petrochenkov M., Stotland I., Mushtakov R. Approaches to Stand-alone Verification of Multicore Multiprocessor Cores. Trudy ISP RAN/Proc.ISP RAS, vol. 28, issue 3, 2016, pp. 161-172. DOI: 10.15514/ISPRAS-2016-28(3)-10.

[12] Kelton W., Law A. Imitatsionnoe modelirovanie [Simulation modeling] // Klassika CS. 3-e izd. SPb.: Piter, 2004.

[13] TLM-2.0.1. TLM Transaction-Level Modeling Library. http://www.accellera.org/downloads/standards/systemc (12.04.2020).

[14] 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language https://standards.ieee.org/standard/1800-2017.html (12.04.2020).