# UVM-based verification assembly and DUT writing

**Abstract**

In the field of chip SoC verification, UVM [3] (Universal Verification Methodology) verification methodology has become the most common and reliable verification method due to its high standardization, scalability and reusability. UVM verification methodology can ensure a more comprehensive verification of DUT, so writing reliable verification components is a key step in the verification process, because these components constitute the core of the verification environment and directly affect the quality and efficiency of verification. Through carefully designed verification components, various test scenarios can be simulated to ensure that the behavior of DUT (Design Under Test) under different conditions meets expectations. In addition, UVM's object-oriented characteristics and componentization strategy make the construction of the verification environment more modular, easy to manage and maintain, and also support complex test sequence generation and execution, further improving the coverage and accuracy of verification. Therefore, UVM verification methodology can not only help us to conduct a comprehensive verification of DUT, but also improve the efficiency of verification, reduce costs, and ensure the quality and reliability of chip design.
**Keywords:** UVM Verification Methodology, Verify Components, DUT.

## 1 Introduction

In the digital age, every advancement in semiconductor technology is pushing the boundaries of computing power. As the complexity of SoC design continues to rise, traditional verification methods have been unable to meet the growing design and verification challenges [6]. In order to meet this challenge, UVM verification methodology came into being, with its standardized framework and object-oriented design philosophy, providing a powerful and flexible platform for SoC verification [2]. UVM not only improves the efficiency and coverage of verification, but also reduces verification costs and accelerates the transformation of products from design to market.

In this project, we focus on the development of AxiToChi components, which is a key interface that enables the AXI protocol to seamlessly connect with the CHI protocol, thus playing an important role in the field of high-performance computing and data processing. AXI, as a widely used interface protocol in SoC, is responsible for achieving high-speed data transmission between the core and the memory; while the CHI protocol is the key technology for connecting the processor core, which directly affects the performance and reliability of the system. Therefore, the development of the AxiToChi component is not only a technical challenge, but also an important improvement in system performance and stability [4].

In order to ensure that AxiToChi components and their related verification components can achieve the expected performance and reliability in actual applications, we use UVM as the verification platform and build

a series of UVM-based verification components. These components include but are not limited to drivers, sequencers, monitors, and environment controllers, which together form a complete verification environment for simulating various operating scenarios to ensure that AxiToChi components can operate stably under different working conditions.

This document will detail the development process of AxiToChi components, as well as the design and implementation of UVM-based verification components. We will explore how to use the advanced features of UVM to improve the automation and coverage of verification, and how to optimize the performance of components through continuous testing and debugging. Through these efforts, we aim to provide a reliable verification solution for SoC designs to meet the increasingly stringent market needs, The verification structure is as follows1.

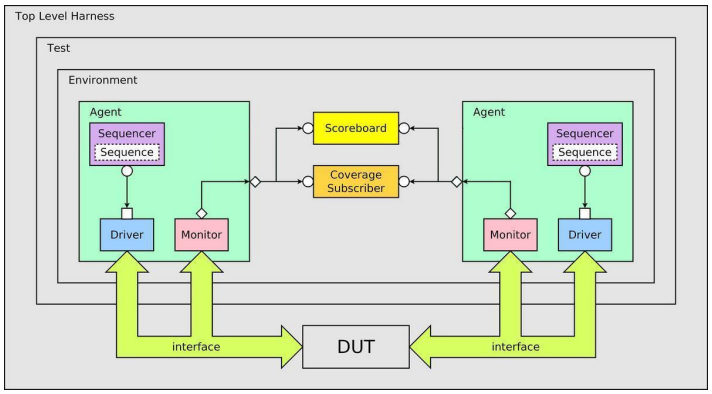



Figure 1. UVM Architecture Diagram.

## 2 Experimental procedures

### DUT coding

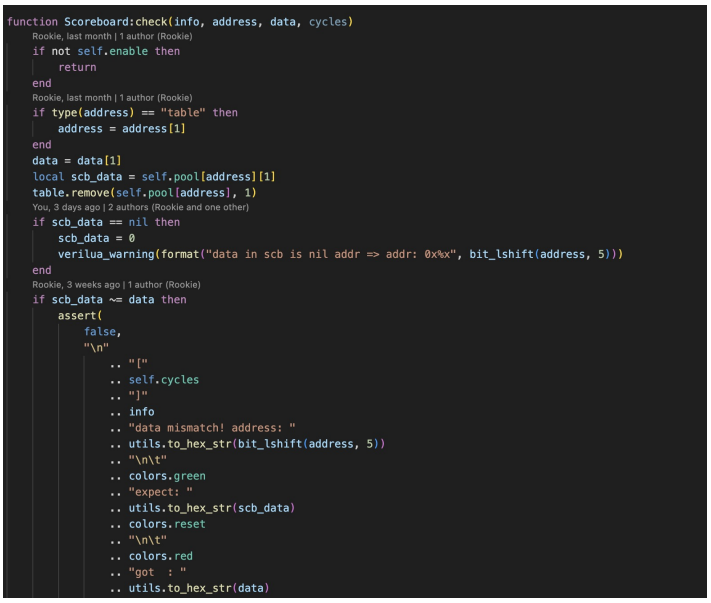
The first step is to write the DUT. I use Chisel to write it. Chisel is a hardware construction language based on Scala. Compared with Verilog, it provides higher code readability and maintainability [7]. It uses the characteristics of modern programming languages to achieve parameterized and constraint-controlled circuit generation, and supports high-level abstraction and functional programming, thereby improving design efficiency and performance. In this design, an AxiToChi transfer bridge is implemented. Its main function is to convert Arm's AXI protocol into Arm's CHI protocol. The AXI protocol is usually used for peripheral interfaces, while the CHI protocol is mainly used for communication between caches. Therefore, the development of this transfer bridge is of great significance to system design, and there is currently no open source code available, which fills the gap in this field.

### Verify component writing

In this experiment, our subject verification work focuses on the development of two core verification components: ScoreBoard and Monitor. These two components play a vital role in the UVM (Universal Verification Methodology) verification methodology. Their writing and implementation are directly related to the accuracy

and efficiency of the verification environment. Therefore, we have invested great attention and energy in the development of these two components.

ScoreBoard is a key component in the verification environment. It is responsible for storing expected results and comparing them with the actual results passed by the Monitor component to verify whether the behavior of the DUT (Design Under Test) meets expectations. The design and implementation of ScoreBoard needs to accurately simulate the functions of the DUT so that the output of the DUT can be accurately verified. In our experiment, ScoreBoard was written in Lua language, because Lua language is known for its lightweight, flexible and efficient performance, and is particularly suitable for rapid development and prototype verification [1]. Using Lua to write ScoreBoard, we can process data and instructions more efficiently and achieve rapid iteration and verification, ScoreBoard code is follow 2.

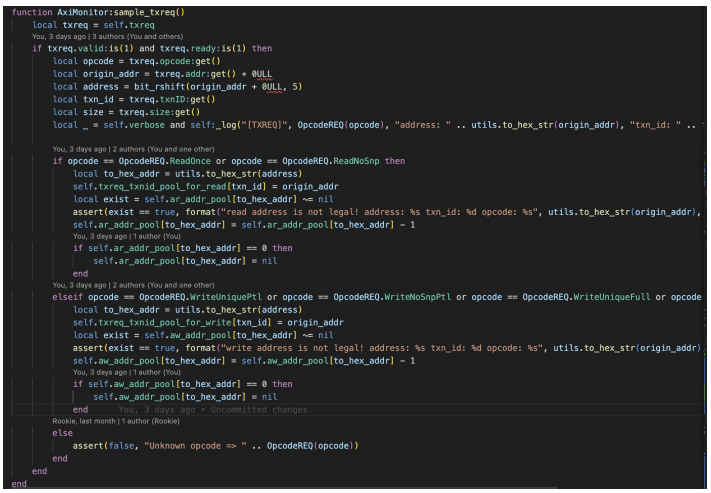


```lua
function Scoreboard:check(info, address, data, cycles)
    Rookie, last month | 1 author (Rookie)
    if not self.enable then
        return
    end
    Rookie, last month | 1 author (Rookie)
    if type(address) == "table" then
        address = address[1]
    end
    data = data[1]
    local scb_data = self.pool[address][1]
    table.remove(self.pool[address], 1)
    You, 3 days ago | 2 authors (Rookie and one other)
    if scb_data == nil then
        scb_data = 0
        verilua_warning(format("data in scb is nil addr => addr: 0x%x", bit_lshift(address, 5)))
    end
    Rookie, 3 weeks ago | 1 author (Rookie)
    if scb_data ~= data then
        assert(
            false,
            "\n"
                .. "["
                .. self.cycles
                .. "]"
                .. info
                .. "data mismatch! address: "
                .. utils.to_hex_str(bit_lshift(address, 5))
                .. "\n\t"
                .. colors.green
                .. "expect: "
                .. utils.to_hex_str(scb_data)
                .. colors.reset
                .. "\n\t"
                .. colors.red
                .. "got   : "
                .. utils.to_hex_str(data)
```

Figure 2. the code of ScoreBoard.

The main responsibility of the Monitor component is to detect the activity of the DUT port in real time and capture and record the behavior of the DUT. It extracts transaction-level information from the DUT and passes it to ScoreBoard for further analysis and verification. In our experiment, the Monitor component is also written in Lua. With Lua's efficient data processing capabilities, Monitor can quickly respond to changes in the DUT and capture and transmit data in a timely and accurate manner. The design of the Monitor needs to take into account the clock domain of the signal to ensure that all monitored signals are in the same clock domain as much as possible, so that ScoreBoard can compare and process them. In addition, the sampling techniques of the Monitor also need to be carefully designed to ensure accurate data capture in different clock cycles and avoid data acquisition errors caused by clock deviation [5], Monitor code is follow 3.

By using Lua to write these two key components, we can make full use of Lua's efficiency and flexibility to improve the response speed and data processing capabilities of the verification environment. This choice not only meets the performance requirements of modern verification methodology, but also reflects our careful consideration of technology selection in verification practice. With the carefully designed ScoreBoard and Monitor, we can verify the DUT more reliably and ensure its correctness and stability under various operating conditions.



```
function AxiMonitor:sample_txreq()
    local txreq = self.txreq
    You, 3 days ago | 3 authors (You and others)
    if txreq.valid:is(1) and txreq.ready:is(1) then
        local opcode = txreq.opcode:get()
        local origin_addr = txreq.addr:get() + 0ULL
        local address = bit_rshift(origin_addr + 0ULL, 5)
        local txn_id = txreq.txnID:get()
        local size = txreq.size:get()
        local _ = self.verbose and self:_log("[TXREQ]", OpcodeREQ(opcode), "address: " .. utils.to_hex_str(origin_addr), "txn_id: " ..

        You, 3 days ago | 2 authors (You and one other)
        if opcode == OpcodeREQ.ReadOnce or opcode == OpcodeREQ.ReadNoSnp then
            local to_hex_addr = utils.to_hex_str(address)
            self.txreq_txnid_pool_for_read[txn_id] = origin_addr
            local exist = self.ar_addr_pool[to_hex_addr] ~= nil
            assert(exist == true, format("read address is not legal! address: %s txn_id: %d opcode: %s", utils.to_hex_str(origin_addr),
            self.ar_addr_pool[to_hex_addr] = self.ar_addr_pool[to_hex_addr] - 1
            You, 3 days ago | 1 author (You)
            if self.ar_addr_pool[to_hex_addr] == 0 then
                self.ar_addr_pool[to_hex_addr] = nil
            end
        You, 3 days ago | 2 authors (You and one other)
        elseif opcode == OpcodeREQ.WriteUniquePtl or opcode == OpcodeREQ.WriteNoSnpPtl or opcode == OpcodeREQ.WriteUniqueFull or opcode
            local to_hex_addr = utils.to_hex_str(address)
            self.txreq_txnid_pool_for_write[txn_id] = origin_addr
            local exist = self.aw_addr_pool[to_hex_addr] ~= nil
            assert(exist == true, format("write address is not legal! address: %s txn_id: %d opcode: %s", utils.to_hex_str(origin_addr),
            self.aw_addr_pool[to_hex_addr] = self.aw_addr_pool[to_hex_addr] - 1
            You, 3 days ago | 1 author (You)
            if self.aw_addr_pool[to_hex_addr] == 0 then
                self.aw_addr_pool[to_hex_addr] = nil
            end       You, 3 days ago • Uncommitted changes
        Rookie, last month | 1 author (Rookie)
        else
            assert(false, "Unknown opcode => " .. OpcodeREQ(opcode))
        end
    end
end
```

Figure 3. the code of Monitor.

## 3 DUT Details

In the design of DUT, Chisel is used and the state machine is used to accurately process AXI requests, and debug can also be easily performed through the state machine. The design of the state machine is a common method in hardware description language, which is used to simulate the timing logic of the circuit to ensure that the data is processed in the correct order at the correct time. The state machine table is as follows 1

Table 1. This is an example table.

| state | conversion conditions | function |
|---|---|---|
| Free | Ready to receive request | Indicates that this Entry is not used |
| SendWrReq | Complete sending Chi request | ready to send request |
| WaitDBID | Receive DBID | Prepare to receive DBID |
| SendWrData | data have been sent | Ready to send data |
| SendCompAck | Response have been sent | Ready to send response |
| Complete | All work have been done | No work to be do |

## 4 Verification Process

After completing the writing of the verification components, we conducted a comprehensive verification of the DUT, the main purpose of which was to ensure that the signals at the DUT ports strictly followed the protocol rules, check whether the addresses converted by the DUT ports were accurate, and verify whether the data transmitted by the DUT ports was correct. In addition, we also needed to ensure that the data was passed to ScoreBoard for update or detection at the appropriate time, which is a critical step in the verification process because it allows us to monitor the behavior of the DUT in real time and compare it with the expected results. If any errors are found during the verification process, the simulation will be interrupted immediately so that we can quickly locate the problem and make corrections to ensure that the DUT can meet the design specifications and performance requirements before actual deployment. This process is critical to improving the reliability of

the DUT and reducing the cost of later corrections.The following figure 4 shows the verification process.

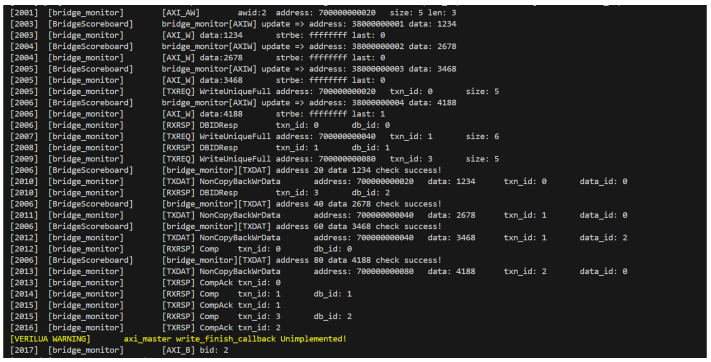



Figure 4. Verification process information

## References


[1] Clifford E Cummings. Ovm/uvm scoreboards-fundamental architectures. *Sunburst Design World Class Verilog, SystemVerilog & OVM/UVM Training*, 2013.

[2] Juan Francesconi, J. Agustin Rodriguez, and Pedro M. Julián. Uvm based testbench architecture for unit verification. In *2014 Argentine Conference on Micro-Nanoelectronics, Technology and Applications (EAMTA)*, pages 89–94, 2014.

[3] Mark Glasser. Configuration in uvm: The missing manual. In *Design & Verification Conference*, 2014.

[4] Saurav Gorai, Saptarshi Biswas, Lovleen Bhatia, Praveen Tiwari, and Raj S. Mitra. Directed-simulation assisted formal verification of serial protocol and bridge. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, page 731–736, New York, NY, USA, 2006. Association for Computing Machinery.

[5] Jeya Prakash Kadambarajan, Pandiaraj Kadarkarai, Budda Kalyani, Mohammad Rabiya Bathul, Theerdhala Sandhya, and Malasani Greeshma. Spi verification monitor module using uvm. In *2024 10th International Conference on Advanced Computing and Communication Systems (ICACCS)*, volume 1, pages 1838–1843. IEEE, 2024.

[6] Peter Wang. A unique centralized-management methodology block/architecture and a novel random input stimulus controlled variable table implementation for the latest marvell ethernet phy uvm verification platform. In *2017 2nd IEEE International Conference on Integrated Circuits and Microsystems (ICICM)*, pages 299–303, 2017.

[7] Mufan Xiang, Yongjian Li, and Yongxin Zhao. Chiselfv: A formal verification framework for chisel. In *2023 Design, Automation  Test in Europe Conference  Exhibition (DATE)*, pages 1–6, April 2023.