Transactions on Computational and Scientific Methods | Vol. 4, No. 9, 2024
ISSN: 2998-8780
https://pspress.org/index.php/tcsm
Pinnacle Science Press

Design and Verification of FlexSPI Controller Using UVM for Efficient Flash Access

Lysandra Quill[1], Carys Imogen[2], Ivo Casimir[3]
[1]Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, USA
[2]Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, USA
[3]Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, USA

Correspondence should be addressed to Lysandra Quill1;lysandra.quill470o@pitt.edu

Abstract: The Serial Peripheral Interface (SPI) is widely used in System-on-Chip (SoC) designs to connect peripherals such as Flash memory due to its simplicity, flexibility, and reliability. As data transmission demands increase, SPI has evolved into advanced standards such as Dual-SPI, Quad-SPI, and the latest xSPI. To bridge the gap between system buses and Flash device interfaces, a FlexSPI controller is required to ensure efficient communication. This paper presents the design and verification of a FlexSPI controller using the Universal Verification Methodology (UVM). The UVM-based verification environment enables rapid development and reuse, addressing the challenges posed by increasing chip complexity and verification costs. The proposed verification platform significantly improves simulation speed, portability, and reusability, providing a valuable reference for future bus module and interface verification tasks.

Keywords: Flex SPI Controller; UVM; Script; Verification.

1. Introduction

Due to its simple structure, flexible configuration and strong reliability, SPI is often integrated in SoC systems to connect peripheral devices such as Flash [1]. With the higher and higher requirements for data transmission rate and data throughput, SPI has also developed from the original standard SPI to Dual-SPI, Quad-SPI, OSPI and the latest xSPI for SPI NOR Flash approved by the international specification organization JEDEC. In most cases, there are differences between the system bus and the interface of the Flash device, and it is necessary to design a corresponding interface protocol conversion circuit [2], that is, the Flash controller. The controller acts as a communication bridge between the core and peripherals in the system, and its function often determines the efficiency of Flash access.

On the other hand, with the increasing complexity and integration of chip design, the difficulty and cost of chip verification are also increasing. The traditional verification method of writing directional use cases through Verilog and manual verification can no longer meet the needs [3]. How to realize the completeness and efficiency of verification in a short period of time has become the goal pursued by various companies. With its high flexibility, reusability and portability, UVM has become the preferred verification methodology in the field of chip design [4] . According to different verification requirements, verifiers can not only quickly build a verification platform suitable for their own needs under the existing framework and powerful components of UVM, but also abstract the environment model and functional modules based on this , is defined as a reusable general verification VIP , which facilitates the reuse of later verification platforms. The introduction of multiplexing technology shortens the development cycle and improves verification efficiency.

2. Research and Function Realization of FlexSPI Controller

As a slave on the AXI subsystem, the FlexSPI controller can convert AXI and APB protocols to SPI protocols, and supports interrupt configuration, software reset, transmission length control, standard SPI transmission mode, and four-wire transmission mode. Access to external serial Flash under different working modes and different transmission rates.

2.1 Top-level Scheme Design of FlexSPI Controller

The core function of the FlexSPI controller is to act as a communication bridge between the core and peripherals. Adopt the top-down design principle, divide different modules and complete their RTL logic coding according to the different functions. The top-level structure is shown in Figure 1, which mainly includes bus slave interface, register configuration, data storage FIFO, and communication control.

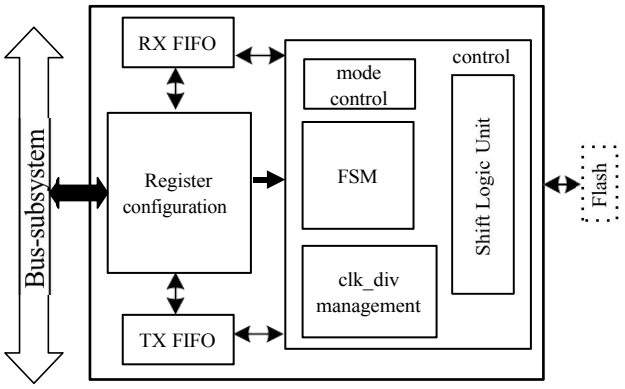


Figure 1. FlexSPI controller top-level structure

2.2 Register Configuration Module

The register configuration module matches the internal registers of FlexSPI by parsing the bus address. FlexSPI has 10 built-in registers, which are STATUS, CLKDIV, CMD, ADDR, LEN, DUM, TX FIFO, RX FIFO, INTCFG, INTSTA. During data communication, registers need to be configured accordingly, and their corresponding functions can be realized by reading and writing control of different registers. The FlexSPI internal registers and their function descriptions are briefly listed in Table 1.

Table 1. FlexSPI Internal Registers Description

REGISTER NAME	ADDR	FUNCTIONAL DESCRIPTION
FLEXSPI_STATUS	0x00	Pass the configuration to the slave, including chip selection, reset, four-wire read-write mode, standard read-write mode
FLEXSPI_CLKDIV	0x04	Used to divide the system clock for FlexSPI transfers to generate the required clock
FLEXSPI_CMD	0x08	The command corresponding to the read or write transfer
FLEXSPI_ADR	0x0C	The address to which a read or write transfer is performed
FLEXSPI_LEN	0x10	Control the length of data sent by the command register, address register and data register
FLEXSPI_DUM	0x14	Dummy cycles between command+address and read/write data
FLEXSPI_TXFIFO	0x18	Write data to TX FIFO
FLEXSPI_RXFIFO	0x20	Read data from RX FIFO
FLEXSPI_INTCFG	0x24	Including interrupt enable, the threshold of triggering interrupt
FLEXSPI_INTSTA	0x28	Interrupt judgment flag, 0: Interrupt failed 1: Interrupt successful

2.3 Communication Control Module

The Control module completes the main functions of the FlexSPI controller, including the implementation of mode selection control, command sequence state machine, clock frequency division management, and shift logic control. FlexSPI completes an access in the form of sending commands, addresses, alternate bytes, idle cycles, and data. The command sequence state transition is shown in Figure 2 (Note: In the figure, the IDLE stage can jump to any stage except WAIT_EG). Any stage in the command sequence can be operated, skipped or executed by configuring the corresponding registers, but at least one of the instruction, address, alternate byte or data stages must be included [5].

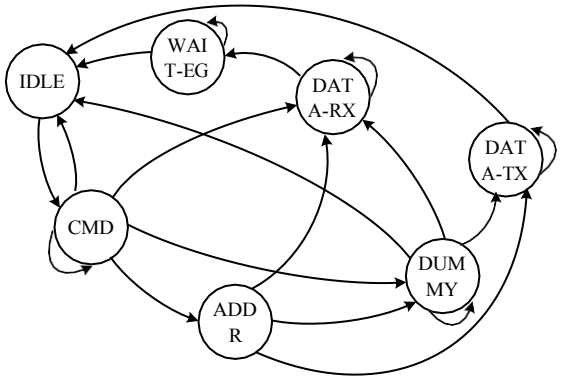


Figure 2. FlexSPI Controller Command Sequence State Transfer

2.4 Clock Divider Management Module

The clock frequency division management module is used to divide the frequency of the system clock transmitted by FlexSPI, which is the output signal of FlexSPI, and provides the clock to the external flash to ensure that the data transmission is carried out according to the flash interface protocol. Realized by a dedicated frequency division counter. The frequency division coefficient can be configured through the register, 8bit value, which can meet different frequency requirements. The clock frequency is given by Equation (1).

$$F\text{lex}SPI_CLK \quad \frac{clk}{2*(CLKDIV \quad 1)} \tag{1}$$

2.5 Shift Logic Control Unit, FIFO Module

The main function of the shift logic control unit is to complete the serial-to-parallel conversion during data communication. The FIFO is mainly used to store data to ensure the continuity of data transmission. As shown in Figure 3: the control unit receives the control information from the controller, and controls the shift logic unit to work. When shifting data, it is necessary to count the number of times of shifting, to judge when to end, and to latch the data at the same time.

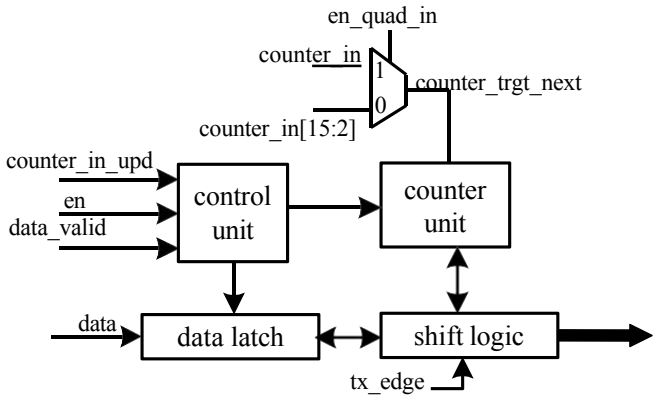


Figure 3. Block Diagram of Shift Logic Control

3. UVM Verification Platform for FlexSPI

In the previous chapters, each module of the Flexspi controller was introduced and analyzed in detail, and on this basis, each module was coded in RTL based on System verilog. The RTL design at this time does not guarantee that the designed functions can be implemented correctly. After a design is realized, we need to go through a lot of verifications to ensure that the design is correct. Therefore, in order to accurately and truly reflect the correctness of the design functions, it is essential to conduct complete and comprehensive verification of the design. A mature and systematic verification process includes the formulation of verification plans and verification schemes, the decomposition of test points, the development of verification components, environment integration, environment operation and simulation analysis, and collection of coverage. The UVM verification platform for the FlexSPI controller will also be built in accordance with this process.

3.1 Verification Plan and Test Point Decomposition

The verification plan defines the content that needs to be verified in the design to be tested, the characteristics of the system, and the necessary verification platform, environment, method, test case and expected result to support the verification process [6]. Test point decomposition is the process of analyzing each function of the design under test one by one with clear semantics, clear structure, and unambiguous descriptive language. The test point decomposition serves for the later verification sequence and test cases. Each test point can Cover with one or more test cases.

The FlexSPI controller serves as a communication bridge between the core and peripherals. The goal of the verification is that the controller can correctly implement all the defined functions. According to the analysis of the characteristics of the FlexSPI controller, Table 2 lists the FlexSPI controller that needs to be verified. Functional features, due to limited space, not all of the more detailed feature divisions are listed.

Table 2. FlexSPI Controller Features

FEATURE	DESCRIBE
Reset	Support hardware reset and software reset
Read-Write mode	Supports standard SPI mode and four-wire transfer mode
Clock divider	Support data communication under different clock
Transfer length	Supports configuration of different transfer lengths and cross-combination
Dummy cycle	Support for adding read-write idle cycles and zero idle cycles
Interrupt	Support interrupt request

3.2 Component Development of Verification Platform

The development of verification components is based on the formulation of the previous verification plan, the realization of the verification scheme and the understanding of the design under test. In this paper, the design under test is a subsystem composed of the FlexSPI controller, axi, and axi2apb, using the 32-bit wide axi4 bus as a bridge interconnection. Therefore, the main components included in the verification environment are transaction agents axi4_agent and flexspi_agent, transaction class, interface class, configuration class config, register model reg_model, scoreboard scoreboard and container class env. TLM communication is used between components. Each component cooperates with each other and is independent of each other, which fully embodies the characteristics of high cohesion and low coupling. At the same time, the bus function model (bfm) and the converter (converter) for data storage and conversion are integrated internally. The function used for data conversion is defined inside the converter, which is used to convert the information in the class defined at the software level and the structure defined at the hardware level, thereby increasing the readability of the data. The role of bfm is to encapsulate the timing of the low-level bus, and provide

a call interface to the top layer, so that the top layer does not need to care about the implementation details of the low layer, and focuses on the generation of incentives and the design of verification cases. This is similar to the object-oriented concept in C++. Objects are equivalent to commands or calls, and the member functions of objects implement specific functions. The outside world does not need to care about the internal details of the class, but only needs to know how to call them. The purpose of bfm is to make the simulation of the verification code faster, the behavior modeling easier, and the model easier to use.

3.3 Environment Integration of the Verification Platform

The UVM-based verification platform is shown in Figure 4. The top layer (top_tb) is composed of test environment components and the design under test. They communicate data through the bus function model bfm and interface. The entire verification platform is divided into two parts according to whether it includes timing and whether it can be synthesized: the timing and synthesizable part is composed of the FlexSPI controller, the subsystem composed of axi and axi2apb, the bus function model (BFM) and the interface. The non-timing and non-synthesizable sections consist of the remaining components of the verification platform, with the ability to speed up simulation and run longer tests.

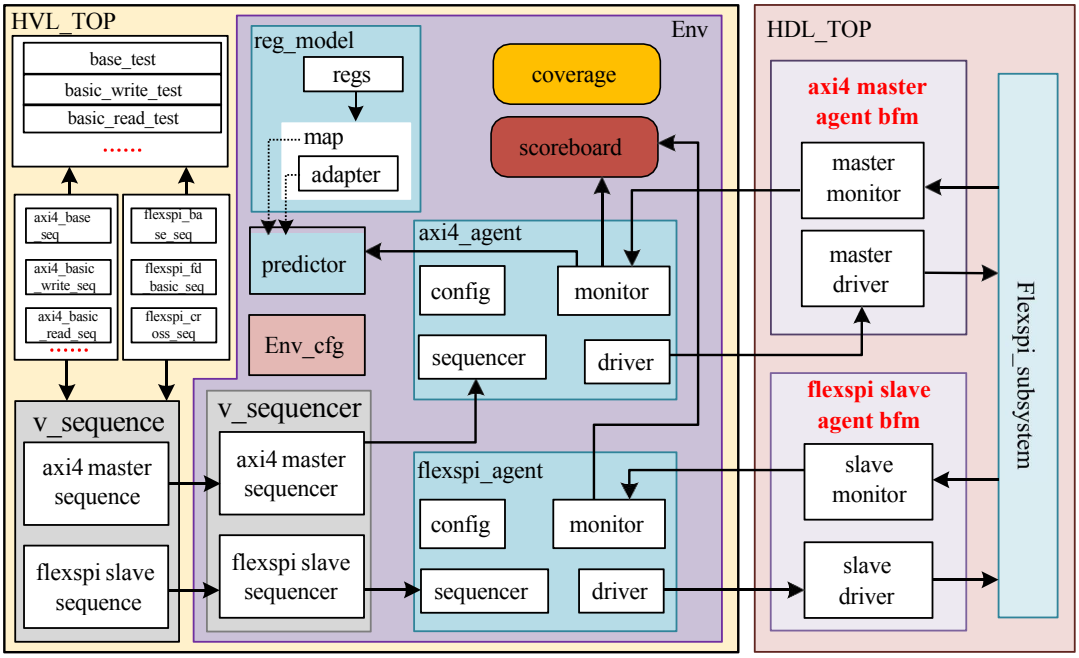


Figure 4. FlexSPI Controller Verification Platform

4. Environment Operation and Simulation Analysis

The operation process of the verification platform in this environment is shown in Figure 5: the top-level top completes the declaration of the interface and the instantiation of the DUT, the generation of the clock and reset signals, and then calls the run_test() function to start the entire verification platform, usually, run_test() is a most basic test case, which contains the common parts of all test cases, the verification platform uses Makefile scripts to design common functions such as compilation, simulation, test case selection, loading waveforms and coverage collection as automated processes [8], the script Add the "+UVM_TESTNAME=$(test)" command to the test parameter to pass different test case names to run different test cases. When all phases in the environment are executed, call the $finsh() function to end the operation of the entire verification platform.

There are two main ways to judge whether the designed function is correct and whether the verification platform meets the construction requirements: one is to check various files output by the verification platform at different stages, such as compilation simulation files and coverage files. The second is to compare waveforms. In the actual verification work, we tend to combine the two methods.

Taking the clock frequency division and basic data transmission functions in the standard mode as an example for analysis, the instructions to be executed are written to each register of FlexSPI through the front door access, and the written instructions, simulation waveforms, and simulation reports are shown in Figure 6-10.

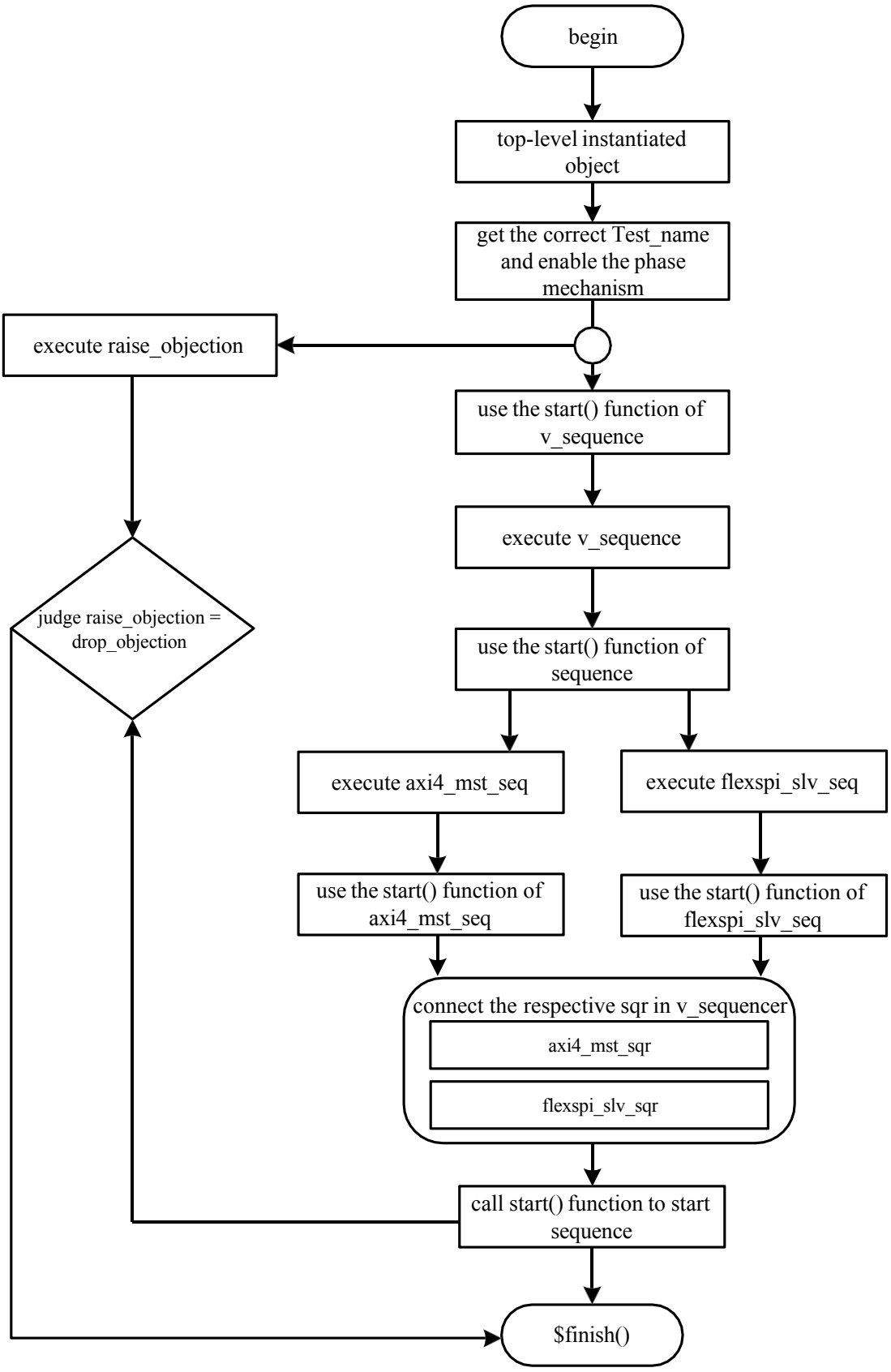


Figure 5. Verify platform operation process

```
#  ------------------------------------------------------------------------------------
# Name                                 Type            Size  Value
# ------------------------------------------------------------------------------------
# axi4_master_tx                       axi4_master_tx  -     @2033
#   tx_type                            string          5     WRITE
#   awid                               string          6     AWID_0
#   awaddr                             integral        32    'h1a102004
#   awlen                              integral        8     'd0
#   awsize                             string          13    WRITE_4_BYTES
#   awburst                            string          10    WRITE_INCR
#   awlock                             string          19    WRITE_NORMAL_ACCESS
#   awcache                            string          16    WRITE_BUFFERABLE
#   awprot                             string          24    WRITE_NORMAL_SECURE_DATA
#   awqos                              integral        4     'h0
#   wait_count_write_address_channel   integral        32    'h0
#   wdata[0]                           integral        32    'h4
#   wait_count_write_data_channel      integral        32    'h0
#   bid                                string          5     BID_0
#   bresp                              string          10    WRITE_OKAY
#   no_of_wait_states                  integral        32    'd0
#   wait_count_write_response_channel  integral        32    'h0
#   transfer_type                      string          14    BLOCKING_WRITE
# ------------------------------------------------------------------------------------
```

Figure 6. FLEXSPI_CLKDIV command

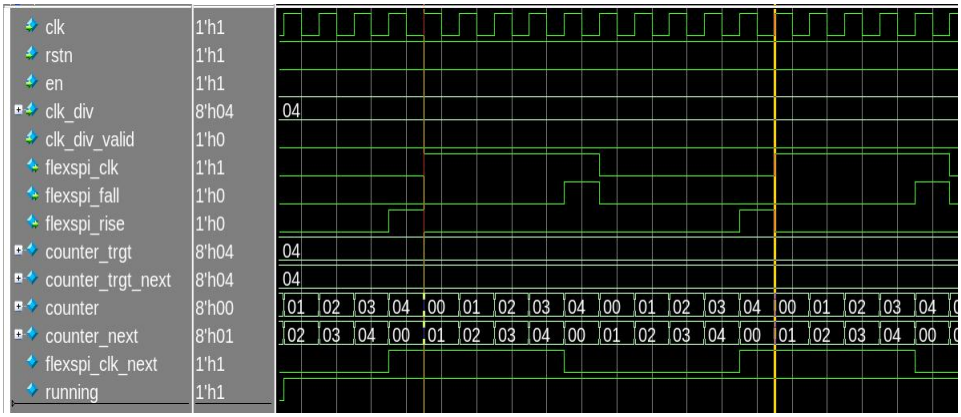


Figure 7. FLEXSPI_CLKDIV simulation waveform

In Figure 6, axi write address awaddr = `h1a102004, corresponding to FlexSPI internal register FLEXSPI_CLKDIV. Write data wdata = `h4, which represents the value of the frequency division coefficient. According to the clock frequency formula (1-1), we can get that the output clock frequency of the controller is one-tenth of the system clock frequency. Analyzing the simulation waveform in Figure 7, we can get It can be seen that the function realization meets the design requirements.

```
#  ------------------------------------------------------------------------------------
# Name                                 Type            Size  Value
# ------------------------------------------------------------------------------------
# axi4_master_tx                       axi4_master_tx  -     @2426
#   tx_type                            string          5     WRITE
#   awid                               string          6     AWID_0
#   awaddr                             integral        32    'h1a102018
#   awlen                              integral        8     'd0
#   awsize                             string          13    WRITE_4_BYTES
#   awburst                            string          10    WRITE_INCR
#   awlock                             string          19    WRITE_NORMAL_ACCESS
#   awcache                            string          16    WRITE_BUFFERABLE
#   awprot                             string          24    WRITE_NORMAL_SECURE_DATA
#   awqos                              integral        4     'h0
#   wait_count_write_address_channel   integral        32    'h0
#   wdata[0]                           integral        32    'hffffff01a
#   wait_count_write_data_channel      integral        32    'h0
#   bid                                string          5     BID_0
#   bresp                              string          10    WRITE_OKAY
#   no_of_wait_states                  integral        32    'd0
#   wait_count_write_response_channel  integral        32    'h0
#   transfer_type                      string          14    BLOCKING_WRITE
# ------------------------------------------------------------------------------------
```

Figure 8. FLEXSPI_TXFIFO command

In Figure 8, axi write address awaddr = `h1a102018, corresponding to FlexSPI internal register FLEXSPI_TXFIFO. Write data wdata = `hffff_f01a, which means write FIFO data. From the simulation waveform in Figure 9, it can be concluded that the write element is 1, the write data data_i = `hffff_f01a, the output element is 1, the output data data_o = `hffff_f01a, and the function implementation meets the design requirements.

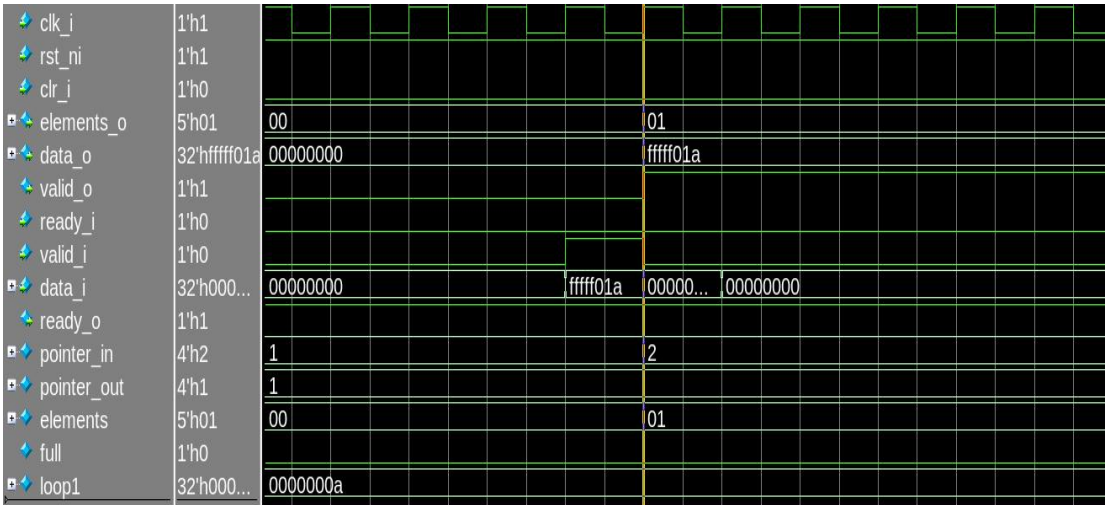


Figure 9. FLEXSPI_TXFIFO simulation waveform

After the verification platform runs, the information of the entire simulation test phase will be counted, and a simulation report similar to that shown in Figure 10 will be generated. The content of the simulation report generally includes platform operation information, data printing information, and test results. Verifiers can also design their own simulation reports according to their needs. In this verification platform, the real-time statistical printing of the coverage rate is added, and the coverage rate of each test case can be intuitively known.

```
# UVM_INFO ../../src/dv/hvl_top/axi4_master_agent//axi4_master_coverage.sv(233) @ 35140:
uvm_test_top.env.axi4_master_agent.axi4_master_cov_h [axi4_master_coverage] AXI4 Master Agent Coverage = 15.75 %
# UVM_INFO ../../src/dv/hvl_top/flexspi_slave_agent//flexspi_slave_coverage.sv(241) @ 35140:
uvm_test_top.env.flexspi_slave_agent_h[0].flexspi_slave_cov_h [flexspi_slave_coverage] flexspi_slave Agent
Coverage = 16.67 %
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  121
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [AXI4_MASTER_DRIVER_BFM]     1
# [MSHA_DEBUG]    18
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [UVMTOP]     1
# [UVM_INFO]     1
# [axi4 master agent bfm]     1
# [axi4_master_coverage]     1
# [axi4_master_driver]    50
# [axi4_master_std_mode_write_even_clkdiv_reg_seq]    22
# [flexspi_fd_basic_slave_seq]     2
# [flexspi_slave_coverage]     1
# [flexspi_slave_driver]     4
# [flexspi_slave_driver_bfm]     5
# [flexspi_slave_monitor]     1
# [flexspi_slave_monitor_bfm]     5
# [std_mode_write_even_clkdiv_reg_test]     4
# ** Note: $finish    : /opt/mentor/questasim10.7/questasim/linux/../verilog_src/uvm-1.1d/src/base/
uvm_root.svh(430)
#    Time: 35140 ns  Iteration: 53  Instance: /hvl_top
# Saving coverage database on exit...
# End time: 20:53:34 on Dec 06,2022, Elapsed time: 0:00:08
```

Figure 10. Simulation report of UVM verification platform

5. Conclusion

In this paper, after analyzing the functional requirements of the FlexSPI controller, the scheme design and RTL coding of all sub-modules are realized. According to its functional characteristics, a detailed verification scheme and verification plan are formulated, and the development of verification components and the integration of the environment are carried out. By designing the functional model, the simulation speed is accelerated. The verification platform designed in this paper has good portability and reusability, and can be used as a reference for building other bus modules and chip interface verification platforms, so as to speed up the construction of the verification platform and improve the verification efficiency.

References

[1] Han Wang, Guangjun Li, Zhiyong Guo. Research and Application of Synchronous Queue Serial Interface QSPI [J]. Single Chip Microcomputer and Embedded System Application, 2008(04):67-69.

[2] Tao Cao. Design and Implementation of Flash Controller on L-DSP Chip [J]. Intelligent Computer and Application, 2020, 10(06):142-147.

[3] Fei Wu. Quad-SPI Controller Design and UVM Verification Based on AHB Protocol [D]. Xidian University, 2021. DOI: 10.27389/d.cnki.gxadu.2021.001273.

[4] Universal Verification Methodology (UVM) 1.1 User's Guide[S]. Accellera, 2011.

[5] Qi Chen, Minhua Ren, Fuzhi Zhang, Biyang Zhang. Design and Verification of Synchronous Serial Interface QSPI [J]. Single-chip Microcomputer and Embedded System Application, 2022,22(08):79-82.

[6] Fiergolski A. Simulation Environment Based on the Universal Verification Methodology[J]. Journal of Instrumentation, 2017,12(1):C01001-C01001.

[7] Yong Guo, Yubo Wang, Xiaoke Tang, et al. A SPI Interface Module Verification Method Based on UVM [C]. 2020 IEEE International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA), Chongqing, China. IEEE, 2020:1219-1223.

[8] Guo, Y. et al.: A SPI interface module verification method based on UVM. In: IEEE International Conference on Information Technology, Big Data and Artificial Intelligence, pp. 1219–1223. Chongqing, China (2020).

[9] Yeqiang Peng. Design and Implementation of QSPI-Flash Controller Applied to Internet of Things Secure Storage [D]. Huazhong University of Science and Technology, 2019. DOI: 10.27157/ d. cnki. ghzku. 2019. 003330.

[10] Zhe Ma. Research and Implementation of APB-SPI Verification IP Based on UVM [D]. Xidian University, 2021. DOI: 10.27389/d.cnki.gxadu.2021.001208.

[11] Zhaobin Li. Research and Implementation of IP Verification of AXI4 Bus Protocol Interface Based on UVM [D]. Jinan University, 2017.