https://doi.org/10.31891/2307-5732-2025-359-28
УДК 004.272.2:004.43:004.054

МАЙКІВ ІГОР
Західноукраїнський національний університет
https://orcid.org/0009-0000-8064-4858
e-mail: imaykiv@yahoo.com
ОСОЛІНСЬКИЙ ОЛЕКСАНДР
Західноукраїнський національний університет
https://orcid.org/0000-0002-0136-395X
e-mail: oso@wunu.edu.ua
БИКОВИЙ ПАВЛО
Західноукраїнський національний університет
https://orcid.org/0000-0002-5705-5702
e-mail: pb@wunu.edu.ua
ФЕДОРОВИЧ ВІКТОР
Західноукраїнський національний університет
https://orcid.org/0009-0009-7806-7751
e-mail: zizix781227@gmail.com

ЗАСТОСУВАННЯ КОНЦЕПЦІЇ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ, ДЛЯ ВЕРИФІКАЦІЇ HDL-ПРОЕКТІВ ЦИФРОВИХ СИСТЕМ

В роботі розглянуто переваги які надає застосування концепції об'єктно-орієнтованого програмування (ООП), реалізованої в мові SystemVerilog, для верифікації HDL-проектів цифрових систем. На прикладі тестбенчу та тестового середовища контролера UART інтерфейсу продемонстровано переваги застосування ООП порівнянні з традиційними підходами. Структура тестового середовища, що базується на застосуванні об'єктів, класів, транзакцій та віртуальних інтерфейсів забезпечує не лише спрощення коду, а також можливість масштабування, розширення та легкої адаптації тестбенчу до інших типів послідовних інтерфейсів, через механізми наслідування та поліморфізму.

Ключові слова: *SystemVerilog, об'єктно-орієнтоване програмування, тестове середовище, тестбенч.*

MAIKIV IHOR, OSOLINSKIY OLEKSANDR, BYKOVYY PAVLO, FEDOROVYCH VIKTOR
West Ukrainian National University

APPLICATION OF OBJECT-ORIENTED PROGRAMMING CONCEPTION FOR VERIFICATION OF HDL-BASED DIGITAL DESIGNS

The article presents the application of an object-oriented methodology for testbench development using SystemVerilog. This approach extends the capabilities of Verilog with complex data types, including classes and principles of object-oriented programming (OOP). The structural and object-oriented approaches are compared in the article. The advantages of OOP are highlighted in the context of code reuse, scalability, and clarity of testbench architecture. The structured verification environment is presented. It is based on transaction-level modeling and incorporates a set of modules (driver, monitor, scoreboard). This methodology allows designers to clearly separate the tasks between different levels and improve maintainability and code reuse.

The practical application of object-oriented approach to testbench design is demonstrated on the UART communication protocol. The developed verification environment supports both transmission and receiving paths (TX and RX), and provides mechanisms for transaction generation, data monitoring and automated result checking. The use of virtual interfaces, mailboxes, and dynamically parameterized transactions ensures that the environment can be easily adapted to different DUT configurations and extended with new functionality. The effectiveness of the proposed approach is validated by implementing and simulating a testbench in a iSIM simulation environment of Xilinx Vivado 2020.2 software package. The obtained results demonstrate the benefits of object-oriented principles in terms of code reusability, modularity and readability that make this approach well-suitable for big and complex verification tasks.

The development of parameterized modules (driver, monitor, control block) that will simplify the verification process of widely used serial interface controllers such as I2C, SPI, QSPI, LIN etc. will be the next step of the research.

Keywords: *SystemVerilog, object-oriented programming, test environment, test-bench.*

Стаття надійшла до редакції / Received 03.10.2025
Прийнята до друку / Accepted 15.11.2025

Вступ

Зростання складності проектів цифрових інтегральних мікросхем (ІМС) веде до значного ускладнення процесу їх проектування, а особливо верифікації [1-5]. За оцінками провідних виробників мікроелектроніки, час який витрачається на верифікацію становить 60-70% від загального часу витраченого на реалізацію проекту [6,7]. Це обумовлено особливостями мов програмування, які традиційно використовуються для створення а також верифікації HDL-проектів. Традиційні мови опису апаратури, такі як Verilog [8,9] та VHDL [8,10], по при їх широке поширення, мають ряд суттєвих обмежень, зокрема в можливості структурованого представлення даних та повторного використання раніше написаного коду в тестбенчі (ТБ), що обумовлює необхідність пошуку нових та більш ефективних підходів до процесу верифікації HDL-проектів.

Традиційні підходи до реалізації та верифікації hdl-проектів

Традиційно, проектування цифрових систем та їх верифікація виконується за допомогою процедурного підходу, реалізованого на основі мов опису апаратури, зокрема Verilog, VHDL [9, 10].

Попри те що Verilog ще у 1995 році став галузевим стандартом і був доопрацьований у 2001 таі 2005

роках, його можливості залишається обмеженими з погляду абстрагування даних, масштабування та повторного використання коду при створенні ТБ.

Verilog, як більшість поширених мов програмування того часу, таких як С є процедурною мовою, синтаксис якої близький до мови програмування С. Однак, на відміну від мови С, де користувач може розширити представлення даних через структури, перелічувані типи, об'єднання та оперувати ними як цілісними об'єктами, Verilog не надає таких можливостей [8,9]. Описуючи проект на мові Verilog розробник оперує обмеженим набором типів даних net (wire) та variable (reg), які дозволяють описати біт, біт-вектор та масиви. Як приклад, коли необхідно зберегти інформацію про складний об'єкт, наприклад пакет даних послідовного інтерфейсу, то виявляється що інформація про транзакцію зберігається в різних масивах, які відображають окремі поля (адреси, даних, контрольної суми тощо). Відсутність можливості оголосити структуру, як новий тип даних який об'єднує необхідні поля, вимагає створення додаткового коду для формування/опрацювання повідомлення при його передаванні/прийманні та перевірки повідомлень, що ускладнює можливість масштабування та повторного використання коду.

Зазвичай, структура ТБ організована навколо статично описаних модулів та сигналів. Взаємодія з модулем, який верифікується (Device Under Test - DUT), відбувається за допомогою initial, always та assign блоків, які не мають чіткого функціонального розділення (генерування, моніторинг, перевірка). Сукупність блоків що оточують DUT та формують необхідну інфраструктуру для проведення верифікації називається тестове середовище (ТС). ТС є невіддільною частиною ТБ. Як наслідок, ТБ має однорангову "плоску ієрархію", жорстко прив'язану до структури DUT, що значно ускладнює його повторне використання в інших проектах або адаптацію до нових конфігурацій [6].

Особливо зростає складність задачі, коли необхідно виконати повну верифікацію функціональних можливостей проекту, наприклад взаємодію через послідовні інтерфейси, обмін даними по системній шині, тощо. В цьому випадку, моделювання виконується на вищому (абстрактному) рівні, де перевіряють алгоритм роботи в цілому, а ТС оперує складнішими об'єктами. У таких випадках, процедурний підхід до створення ТС потребує значних затрат його створення та супровід в процесу верифікації, адже всі зміни вимагають редагуванням існуючої структури ТС. Як приклад, коли виникає необхідність розширити тест, додавши перевірку нових функціональних можливостей, то необхідно відредагувати весь код та виконати повторний синтез, оскільки, всі масиви є статичним. Тому, розмір масивів задають виходячи із максимально можливого числа трансакцій однак, протягом виконання типового тесту, більша частина виділеної пам'яті не використовується. Виходячи із вище сказаного, тестові середовища, створені на основі процедурного підходу, мають ряд принципових обмежень, зокрема:

1) жорстка структура без можливості динамічної зміни у процесі моделювання;
2) низька масштабованість для складних багаторівневих систем;
3) дублювання коду в різних частинах тестового середовища;
4) складність побудови автоматизованих сценаріїв (тест-планів) та їх покриття.

Вище наведені недоліки вказують на необхідність пошуку нових підходів до проектування ТС із врахуванням переваг та нових можливостей які надає мова SystemVerilog [11-16], та сучасних технологій проектування, зокрема запропонована фірмою Xilinx технологія High-Level Synthesis [17].

Формулювання цілей статті

Метою роботи є: пошук нових підходів та рекомендацій до створення ТС які забезпечать: 1) роботу із структурованими типами даних; 2) модульний підхід з можливістю динамічної зміни структури ТС; 3) повторне використання компонентів ТС у нових проектах; 4) можливість розширення верифікації (тест-плану) без необхідності зміни структури ТС.

Виклад основного матеріалу

1. Застосування концепції ООП для створення компонентів тестового середовища

Для вирішення вище означених проблем пов'язаних як із проектуванням а особливо із верифікацією апаратних проектів, фірмою Accellera було запропоновано вдосконалити мову Verilog додавши багато функціональних удосконалень, як в області проектування так і верифікації. Нова мова програмування отримала назву SystemVerilog та затверджена як стандарт IEEE1800-2012 [11]. Зокрема, в SystemVerilog додано нові типи даних які дозволяють розв'язати традиційні проблеми дизайнерів та інженерів які займаються верифікацією.

Принциповою відмінністю SystemVerilog від традиційних Verilog та VHDL є застосування концепції об'єктно-орієнтованого програмування (ООП) для вдосконалення процесу верифікації HDL-проектів [13, 14].

Клас є новим типом даних які декларує розробник (user-defined data type). Такий підхід дозволяє створювати складні типи даних та одночасно пов'язувати їх із процедурами (задачами та функціям) які безпосередньо взаємодіють із ними. Це дозволяє створювати ТБ та системні моделі пристроїв на вищому (абстрактному) рівні, а також виконувати набір операцій викликаючи процедури замість управляння окремими розрядами. Робота на вищому рівні із транзакціями дозволяє спростити код який стає легший для розуміння та зменшити витрати на його розроблення. Такі ТБ безпосередньо не пов'язані із деталями проекту, що робить їх більш надійними та дозволяє повторне використання компонентів ТС в наступних проектах.

Концепція ООП яка реалізована у SystemVerilog базується на базових принципах ООП [18], але в той же час оптимізована для моделювання апаратних засобів. Тому, окрім понять які є традиційними для ООП вводиться ряд нових понять [11]: 1) клас (class) - тип даних оголошений розробником який поєднує в собі дані та методи їх опрацювання, аналогом якого в мові Verilog є модуль; 2) об'єкт (object) – екземпляр класу. В мові

Verilog розробник повинен оголосити екземпляр модуля перед його використанням; 3) вказівник (handle) – вказівник на об'єкт. В мові Verilog розробник повинен вказати ім'я екземпляра коли звертаються до сигналу та підпрограм (routine) які оголошені поза модулем; 4) властивості (property) – змінні оголошені в середині класу, які містять дані, їх аналогами в мові Verilog є сигнали (змінні) оголошені як reg. При створенні класу заборонено оголошувати дані типу net (wire); 5) методи (methods) – задачі (tasks) та функції (functions) що містять процедурний код який опрацьовує змінні. Функції є неблокуючими методами які негайно обчислюють та повертають значення. Їх використовують для обчислення необхідних значень, параметрів тощо. Задачі є блокуючими методами які виконуються в часі, їх використовують для моделювання операції що виконуються в часі, як приклад передавання або приймання пакета даних.

Означений підхід дозволяє будь-яку модель розглядати як об'єкт класу. Як приклад, дані які передаються по послідовному інтерфейсу у вигляді пакетів розглядаються як об'єкти, а їх окремі поля як властивості класу. Методи використовуються для розрахунку та перевірки службових полів таких як контрольна сума або біт парності/не парності, формування із послідовного вхідного потоку даних окремих полів повідомлення, при прийманні та виконання зворотних операцій при передачі. Клас декларується із використанням ключових слів (class-endclass). На рис. 1 наведено приклад оголошення класу frame який описує пакет повідомлення.

```
class frame
    logic [4:0] addr;
    logic [7:0] payload;
    logic parity = 0;

    function new (input int add, dat);
        addr = add;
        payload = dat;
        genpar ();
    endfunction

    function void genpar();
        parity = ^(addr, payload)
    endfunction

    function logic [13:0] getfarme();
        return ({addr, payload, parity});
    endfunction

endclass
```

Рис. 1. Приклад оголошення класу

Наведений приклад класу frame містить три властивості: 1) addr - поле адреси; 2) payload - поле даних; 3) parity - біт парності. Також три методи що оголошені як функції: 1) new – конструктор, створює новий об'єкт класу; 2) genpar - обчислює біт парності; 3) getfarme - повертає всі властивості класу у вигляді 14-розрядного вектора. Один із методів класу завжди є конструктором, який містить ключове слово new та дозволяє створювати об'єкти класу. Конструктор ніколи не повертає значення, лише вказівник (handle) на об'єкт класу. Приклад створення нового об'єкта класу наведено на рис. 2.

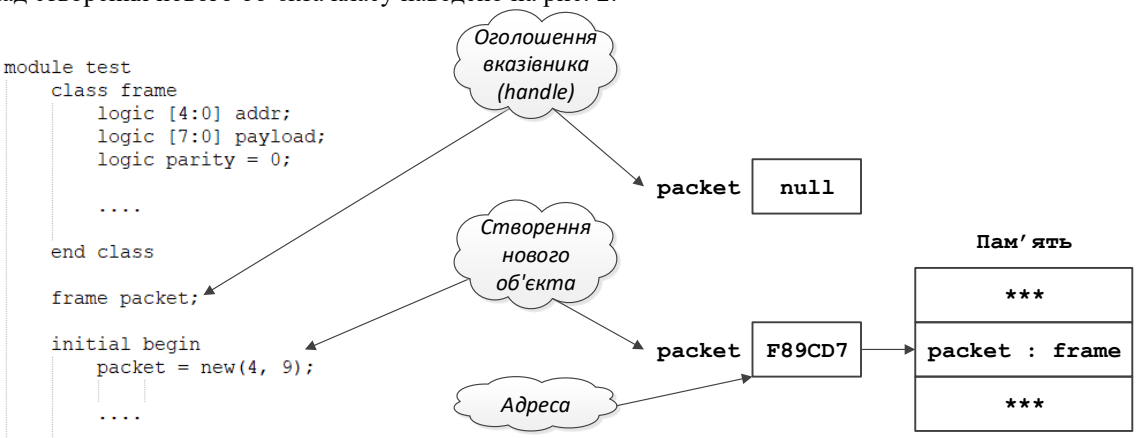

Рис. 2. Приклад створення нового об'єкта класу

Використовуючи клас спершу необхідно оголосити вказівник (handle) який не містить жодного значення (null). При створенні об'єкта викликається конструктор який виділяє певну область пам'яті для об'єкта та записує відповідну адресу у вказівник. В SystemVerilog реалізовано автоматичне керування пам'яттю. Об'єкти автоматично створюються та видаляються в процесі моделювання. Базуючись на принципі наслідування SystemVerilog дозволяє створювати похідні класи. Наслідування класів дозволяє розробнику створювати ряд сценаріїв тестування (тест-планів) залежно від сформульованої множини задач. Користувач

може створити об'єкти різних класів які випадковим чином формують пакети даних та надсилати їх на DUT. На рис. 3 наведено приклад наслідування.

Базуючись на принципі наслідування SystemVerilog дозволяє створювати похідні класи. Наслідування класів дозволяє розробнику створювати ряд сценаріїв тестування (тест-планів) залежно від сформульованої множини задач. Користувач може створити об'єкти різних класів які випадковим чином формують пакети даних та надсилати їх на DUT. На рис. 3 наведено приклад наслідування. На основі базового класу frame розробник оголошує похідний клас tag_frame в який додано дві нові властивості: 1) frame_cnt - вказує загальне число згенерованих пакетів; 2) tag – номер пакета. Також додано два методи: 1) get_frame_cnt - повертає інформацію про загальне число пакетів; 2) get_tag - повертає номер пакета. В процесі верифікації, розробник знатиме загальне число пакетів що поступили на DUT, номер кожного пакету та як він був опрацьований. Якщо один із пакетів викликав хибну роботу DUT, розробник знатиме його номер, що дозволить більш швидко виявити помилку. На основі класу tag_frame створено похідний клас err_frame який формує пакети із помилкою, що дозволяє перевірити як DUT опрацьовує повідомлення які містять помилку. В новому класі додано властивість err_cnt яка вказує загальне число пакетів які містять помилку. Також додано два методи: 1) add_error - додає помилку в повідомлення; 2) get_error - повертає число пакетів із помилкою.

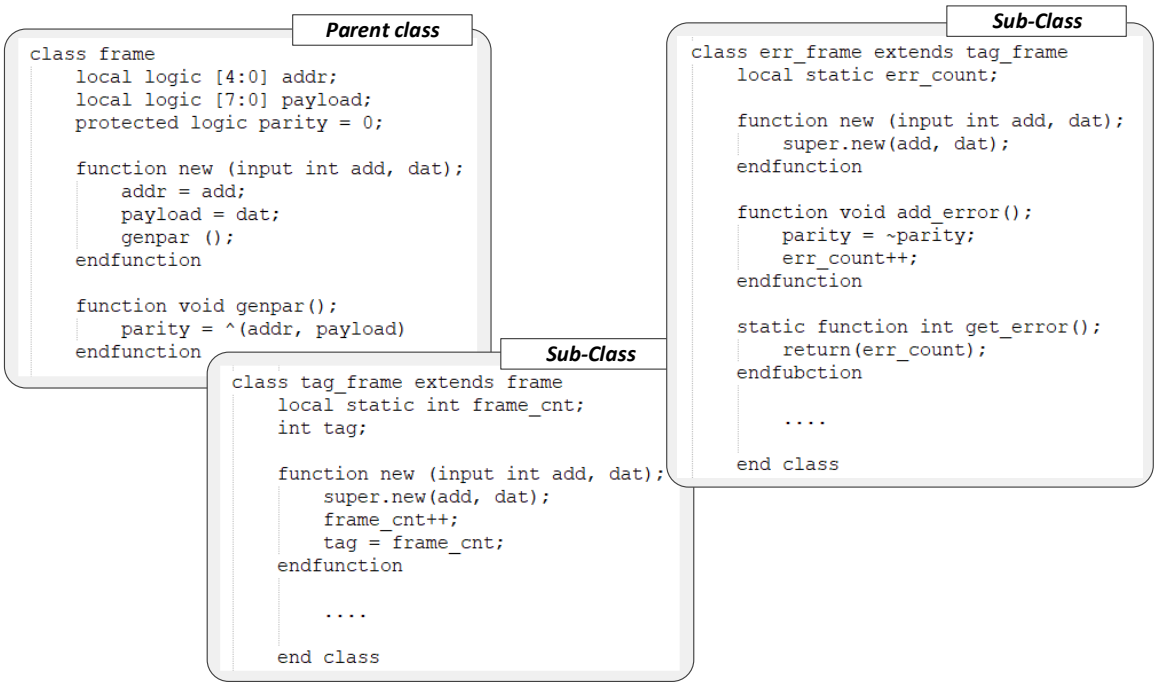

Рис. 3. Наслідування класів

Наслідування дозволяє похідним класам використовувати дані та методи батьківських класів, що зменшує ризик припуститися помилки.. Водночас SystemVerilog не підтримує множинне наслідування

2. Організація тестового середовища на основі концепції ООП

Традиційний підхід до створення ТБ організований навколо операцій які необхідно виконати, а саме: створити трансакцію, надіслати її DUT, прийняти вихідні дані, перевірити їх та сформувати звіт про результати тестування. Verilog та SystemVerilog використовують концепцію оголошення екземплярів, однак вони принципово відрізняються. У Verilog модулі використовуються для оголошення екземплярів компонентів при створенні моделі проекту а також ТБ [6]. Створюючи на мові Verilog ТБ для складного проекту, розробнику необхідно розробити набір необхідних модулів та задекларувати їх відповідно до ієрархії проекту. При цьому, оголошення є статичними що не дозволяє змінювати структур ТБ під час моделювання.

Принципово інший підхід пропонує SystemVerilog, використовуючи концепцію ООП [12-16], Структура ТБ на розділена на дві частини: структурну та область класів (див. рис. 4), а процес верифікації розділено на 4 рівні, де ТС організоване на базі трансакції та операції які виконуються над ними.

Розробник оголошує набір необхідних класів та створює необхідне число об'єктів кожного класу, щоб відтворити необхідну ієрархію ТС, а ТБ генерує об'єкти які формують ТС в процесі верифікації. Такий підхід, за рахунок модульного підходу та ізольованості компонентів, спрощує процес створення ТС та верифікації на кожному із рівнів, дозволяє ефективно організовувати структуру ТС. Всі об'єкти обмінюються між собою лише даними, а опрацювання даних здійснюється в середині кожного об'єкта, що дозволяє легко розширити функціональність ТБ за рахунок наслідування класів та введення нових функцій, без порушення базової структури ТБ. Це що суттєво підвищує ефективність і гнучкість в процесі верифікації HDL-проектів.

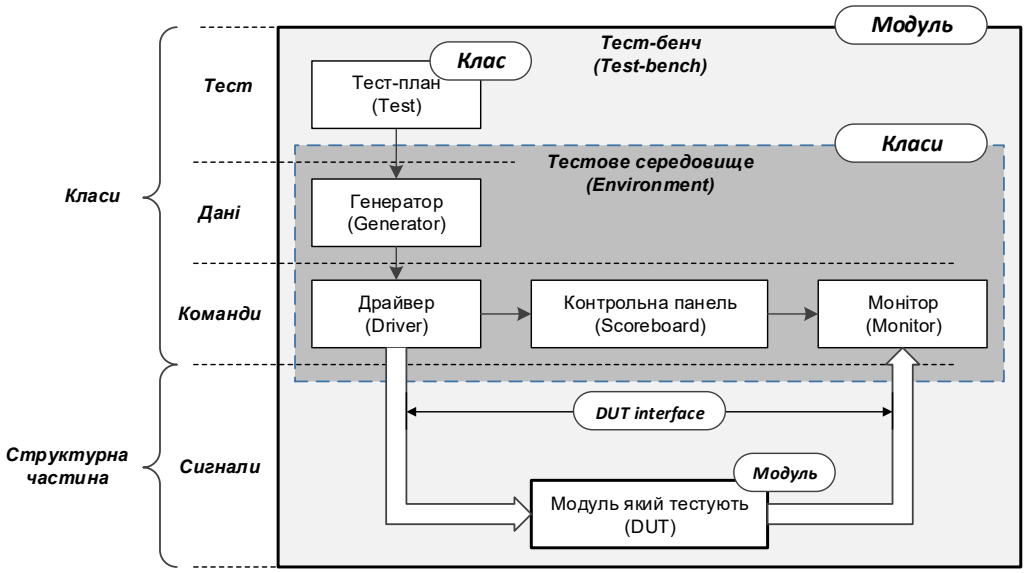

Рис. 4. Рекомендована структура тестбенчу

Високорівнева верифікація [12,-15] передбачає перевірку всіх функціональних можливостей DUT. Для цього розробник розробляє тест-план який складається з окремих тестів кожен із яких забезпечує перевірку DUT. На цьому етапі, розробник вказує який функціонал тестуємо та які дані потрібно згенерувати, не заглиблюючись в деталі пов'язані із формуванням тестових даних, їх передаванням, прийманням відповіді та наступним опрацюванням прийнятих даних. На рівні даних, генератор, отримавши інформацію про те який функціонал буде перевірятись, формує набір тестових даних, пакетів, транзакцій які забезпечують відповідну перевірку та надсилає їх драйверу. На рівні команд відбувається керування сигналами при передаванні та контроль за сигналами при прийманні тестових даних, зокрема: 1) драйвер - забезпечує виконання окремих команд керуючи сигналами на входах DUT, як приклад запис в пам'ять або читання із пам'яті; 2) монітор - контролює вихідні сигнали DUT забезпечує їх групування та конвертує в об'єкт які надсилає у контрольну панель; 3) контрольна панель - порівнює отримані результати із очікуваними та формує звіт про результат тестування. В структурній частині розташовано DUT та набір сигналів які забезпечують його взаємодію із ТС, взаємодія між якими відбувається через інтерфейс, який включає набір всіх сигналів DUT.

Результати синтезу та моделювання

Для оцінки ефективності застосування концепції ООП в процесі верифікації, на мові SystemVerilog реалізовано HDL-проект контролера UART інтерфейсу, із використанням програмного пакета фірми XILINX Vivado 2020.2. Узагальнена структурна схема контролера UART інтерфейсу, отримана після завершення етапу синтезу, наведена на рис. 5, та включає два окремі модулі, приймача (uart_rx) та передавача (uart_tx).

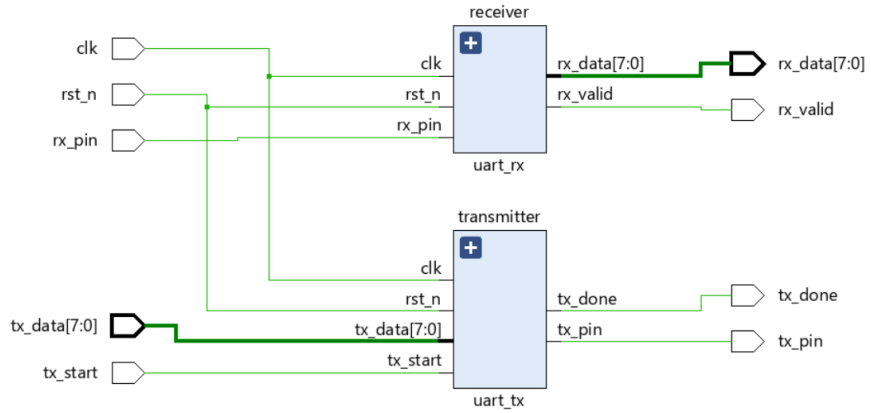

Рис. 5. Узагальнена структурна схема контролера UART інтерфейсу після завершення етапу синтезу

Контролер має наступні входи та виходи: 1) clk – вхід синхронізації; 2) rst_n – вхід скидання; 3) rx_pin – послідовний вхід приймача; 4) rx_data[7:0] – 8-розрядна шина прийнятих даних; 5) rx_valid – вихідний сигнал, який підтверджує успішне завершення приймання даних; 6) tx_data[7:0] – 8-розрядна шина даних які необхідно передати; 7) tx_start – вхід на який надходить сигнал керування по якому розпочинається передавання даних; 8) tx_done - вихідний сигнал який поінформує про завершення передавання даних.

Для верифікації контролера UART інтерфейсу створено ТБ в якому, на основі класів, формується ТС. На рис. 6 наведено набір класів оголошених в ТБ. Для взаємодії між контролером та ТС в ТБ оголошено інтерфейс uart_if. Нижче наведено перелік класів з описом їх даних та методів:

1) uart_transaction – створює об'єкти, які містять дані для передачі та зберігають прийняті дані. Клас містить три змінні: 1) data; 2) tx_done; 3) rx_valid, а також конструктор new ();

2) uart_driver – створює об'єкт який через інтерфейс надсилає дані призначені для передачі у поштову скриньку драйвера (drv_mbx) та контролює процес передавання даних. Клас містить дві змінні: 1) віртуальний інтерфейс vif типу uart_if; 2) поштову скриньку драйвера drv_mbx. Вказані змінні передаються конструктору класу new як аргументи, при створенні нового об'єкту. Основний метод класу run створює об'єкти tx типу uart_transaction та надсилає їх у поштову скриньку драйвера drv_mbx, звідки вони через інтерфейс vif надходять на шину tx_data модуля uart_tx. Одночасно на виході vif.tx_start формується імпульс лог.1, який включає процес послідовної передачі UART повідомлення;

3) uart_monitor – створює об'єкт що контролює процес приймання даних і через інтерфейс отримує від приймача uart_rx прийняті дані та розміщує їх у поштовій скриньці scb_mbx. Клас містить дві змінні: 1) віртуальний інтерфейс vif типу uart_if; 2) поштова скринька контрольної панелі scb_mbx. Вказані змінні передаються конструктору класу new як аргументи, при створенні нового об'єкту;

4) uart_scoreboard – створює об'єкт який перевіряє коректність роботи DUT. Клас містить три змінні: 1) поштову скриньку інформаційної панелі scb_mbx; 2) чергу, в якій розміщені пакети даних, які надходять у драйвер для передачі expected[$]; 3) пакет даних прочитаний із черги exp. Черга та пакет є об'єктами одного класу uart_transaction. Основний метод класу run() порівнює два пакети даних. Перший пакет вичитують із черги expected[$] а другий пакет із поштової скриньки scb_mbx, в яку монітор завантажує пакети, прийняті модулем uart_rx. Якщо передані та прийняті дані аналогічні, то на консоль виводиться повідомлення про успішну транзакцію, в протилежному випадку виводиться повідомлення про помилку;

5) uart_env – створює об'єкт тестового середовища яке через інтерфейс взаємодіє із DUT. Клас містить шість об'єктів (змінних), які описані вище, а саме: 1) драйвер; 2) монітор; 3) контрольна панель; 4) drv_mbx та scb_mbx – дві поштові скриньки; 5) vif – віртуальний uart_if інтерфейс. Конструктор класу new(uart_if vif) отримує як аргумент віртуальний інтерфейс, а також створює вище перераховані об'єкти (vif, drv_mbx, scb_mbx, scoreboard, driver, monitor), які утворюють ТС. Основний метод класу run запускає в паралельне виконання методи run кожного із об'єктів: driver.run(), monitor.run() та scoreboard.run.

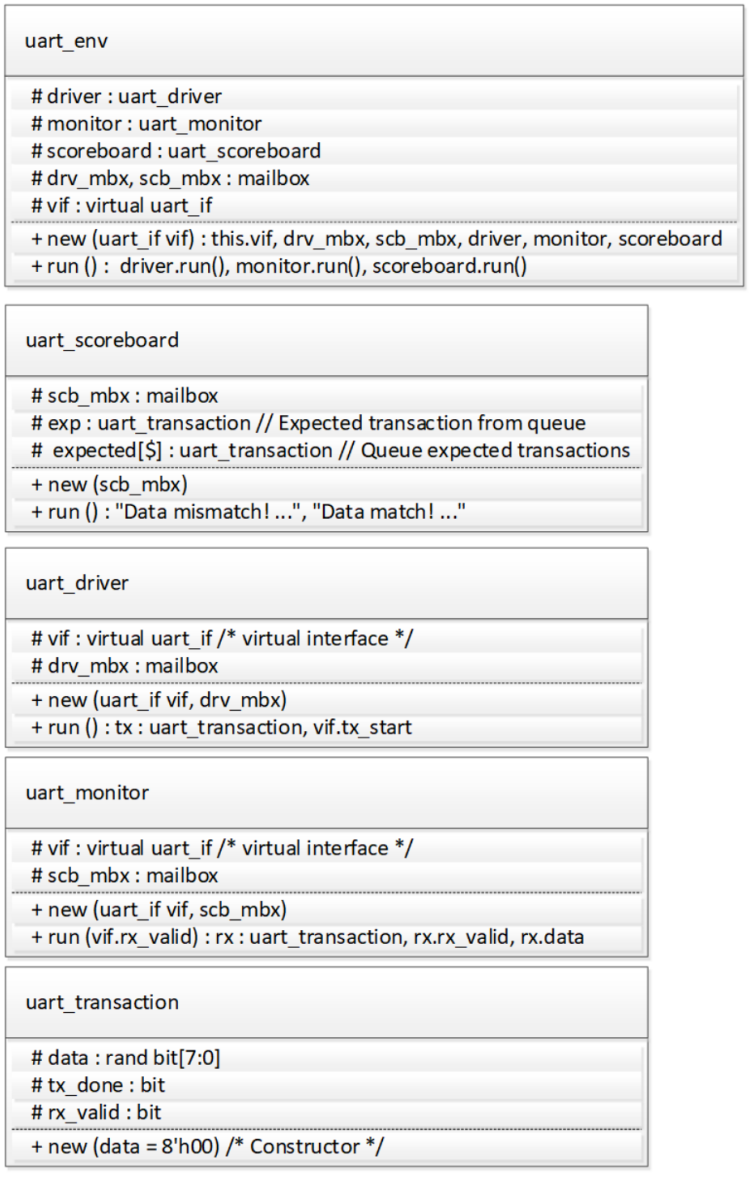

Рис. 6. Перелік класів оголошених у тестбенчі uart_tb

На рис. 7 наведено уривок лістингу декларативної частини ТБ, яка формує його структурну частину.

```
174        // Interface declaration
175        uart_if uart_vif (clk, rst_n);
176        // Declaration of handle (pointer) at object
177        uart_env uart_envirenment;
178        // Instance of devace under test
179        uart dut_uart (
180            .clk      (clk),
181            .rst_n    (rst_n),
182            .tx_start (uart_vif.tx_start),
183            .tx_data  (uart_vif.tx_data),
184            .tx_done  (uart_vif.tx_done),
185            .tx_pin   (uart_vif.tx_pin),
186            .rx_pin   (uart_vif.tx_pin),
187            .rx_data  (uart_vif.rx_data),
188            .rx_valid (uart_vif.rx_valid)
189        );
```

Рис. 7. Лістинг декларативної області тестбенча

Для формування структурної частини ТБ: 1) оголошено віртуальний інтерфейс uart_vif (стріч #175); 2) оголошено вказівника на об'єкт uart_envirenment (стрічка #177); 3) виконано підключення екземпляра модуля контролера UART (стрічки #179-189).

На рис. 8 наведено уривок лістингу виконуваної частини ТБ, яка відповідає за створення ТС та наступне виконання тест-плану.

```
194    initial begin
195        rst_n = 0;
196        #20000 rst_n = 1;
197        // Creating uart_envirenment object
198        uart_envirenment = new(uart_vif);
199        // Running all environment components
200        fork
201            uart_envirenment.run();
202        join_none
203        // Generating transactions via the uart_envirenment API
204        repeat (5) begin
205            uart_transaction tx = new();
206            assert(tx.randomize());
207            $display("TB: Sending 0x%0h", tx.data);
208            uart_envirenment.driver.drv_mbx.put(tx);
209            uart_envirenment.scoreboard.expected.push_back(tx);
210            #1000000 ;
211    end
```

Рис. 8. Частина лістингу тестбенчу яка пояснює роботу тестового середовища uart_envirenment

Процес створення ТС включає два етапи: 1) створюється об'єкт ТС uart_envirenment (стрічка #198) та всі об'єкти які його формують (драйвер, монітор, контрольна панель); 2) використовуючи метод run() відбувається запуск усіх об'єктів ТС (стрічки #200-202).

Далі виконується верифікація модуля, яка передбачає: 1) створення об'єкта трансакції tx 2) присвоєння нових даних об'єкту tx (стрічка #205); 3) розміщення транзакції tx в поштову скриньку драйвера (стрічка #208); 4) розміщення транзакції tx в чергу expected, контрольної панелі (стрічка #209); 5) формування часової затримки необхідної для передавання пакету та опрацювання прийнятих даних (стрічка #209). Вище описаний цикл виконуються 5 раз, і кожного разу передається новий пакет.

Моделювання роботи драйвера UART інтерфейсу виконано у вбудованому симуляторі iSIM програмного пакета Vivado 2020.2. Нижче, на рис.9 наведено структурну частину ТБ яка включає віртуальний інтерфейс uart_vif та модуль який тестується – dut_uart.

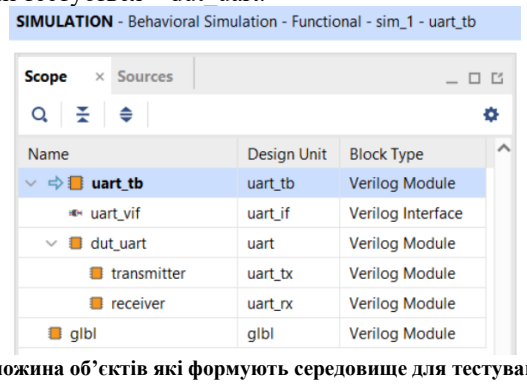

Рис. 9. Множина об'єктів які формують середовище для тестування UART

В процесі верифікації, ТБ створює ТС uart_envirenment яке містить набір об'єктів на основі раніше описаних класів, а саме: драйвер, монітор, контрольну панель, поштові скриньки віртуальний інтерфейс (див. рис.10)

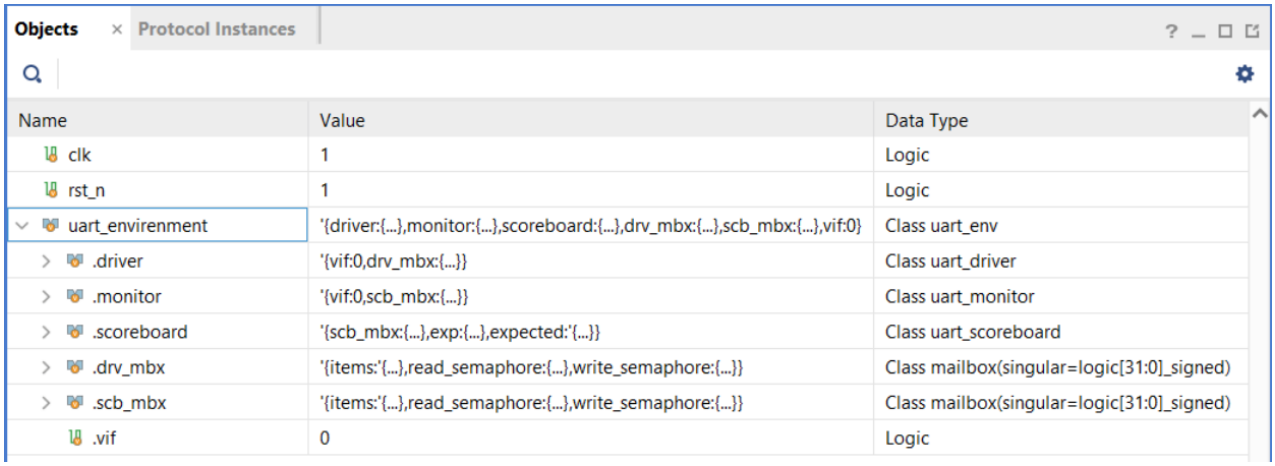

Рис. 10. Тестове середовище uart_environment та множина об'єктів сформованих для тестування UART

Нижче, на рис. 11 наведені часові діаграми отримані при моделюванні роботи драйвера UART інтерфейсу на швидкості передавання даних 115,2 кбіт/с. На представлених часових діаграмах UART передавач, отримавши через віртуальний інтерфейс від драйвера об'єкт tx класу uart_transactions із полем даних tx_data = 0x2d, виконує його передачу через послідовний вихід tx_pin. Одночасно, UART приймач, через послідовний вхід rx_pin, приймає та опрацьовує повідомлення. Прийняті дані, через віртуальний інтерфейс, надходять у монітор. Сигнали tx_done та rx_valid через віртуальний інтерфейс інформують драйвер та монітор про завершення передачі та приймання повідомлень.

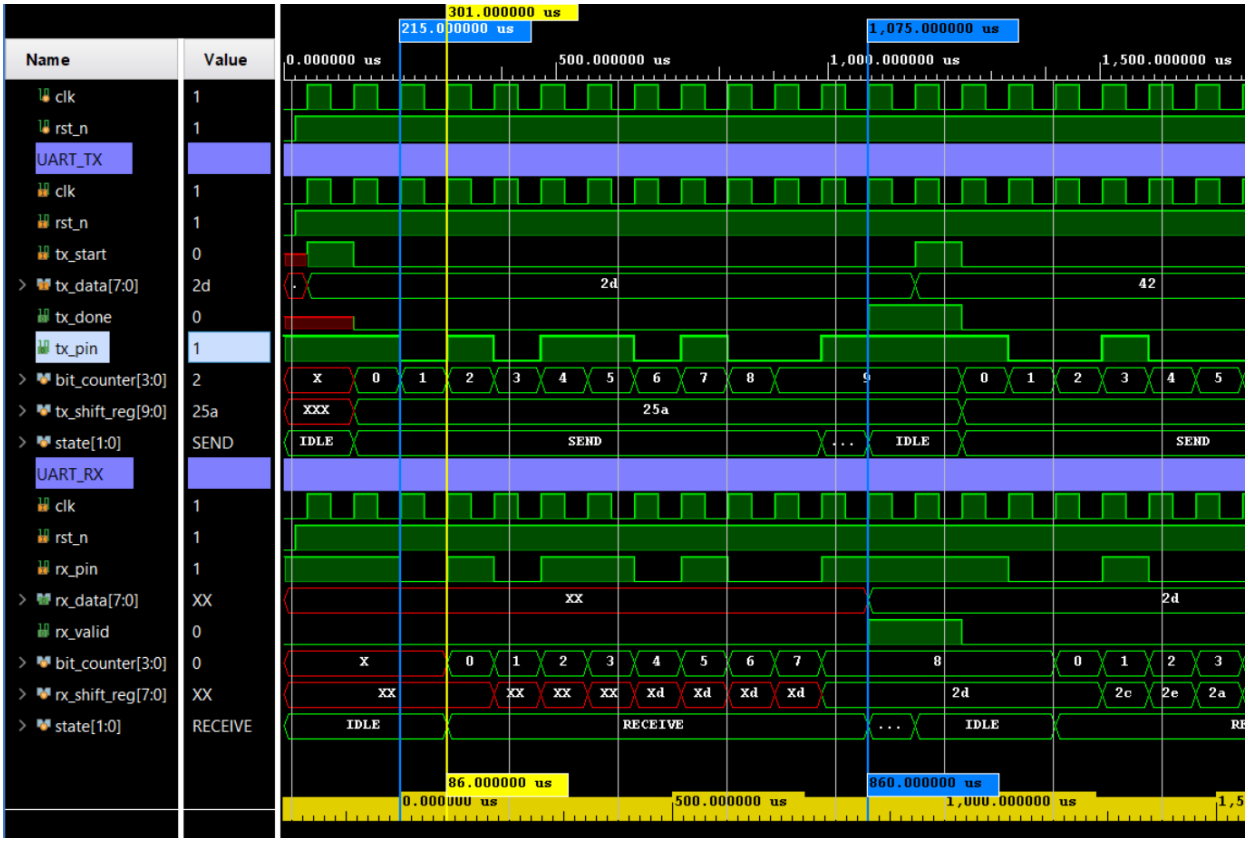

Рис. 11. Часові діаграми роботи приймача (UART_TX) та передавач (UART_RX)

Паралельно із структурною частиною ТБ працює ТС (uart_envirenment), результати роботи якого, у вигляді повідомлень, виводяться на консоль (див. рис. 11).

На рис. 11 наведено декілька етапів роботи тестового середовища, а саме:

1) етап #1 – ТБ створює об'єкт tx=0x2d класу uart_trasaction, який надсилається у ТС, яке ініціює роботу драйвера, монітора та контрольної панелі;

2) етап #2 – ТБ створює та передає новий об'єкт tx=0x42. В той час, компоненти ТС виконують

передавання та приймання транзакцій через віртуальний інтерфейс. Прийнятий об'єкт tx=0x2d надходить в контрольну панель. Остання виконує перевірку, результат якої відображається повідомленням.

```
TB: Sending 0x2d
UART DRIVER: Started
UART MONITOR: Started
UART SCOREBOARD: Started
TB: Sending 0x42
UART DRIVER: Sent data 0x2d
UART MONITOR: Received data 0x2d
SCOREBOARD: Data match! 0x2d
```

Рис. 12. Повідомлення, які відображають роботу тестового середовища uart_envirenment

Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі

За результатами виконаного дослідження проведено аналіз переваг та можливостей які надає застосування концепції ООП у мові SystemVerilog для створення ТБ та верифікації HDL-проектів.

Для оцінки ефективності застосування концепції ООП для верифікації HDL-проектів, на мові SystemVerilog створено проект контролера UART інтерфейсу та ТБ, в якому виконане моделювання та перевірено роботу контролера. Зокрема, в ТБ оголошено множину класів, на основі яких, формується ТС, яке бере під свій контроль процеси моделювання та верифікації отриманих результатів. Практичне моделювання підтвердило коректну роботу всіх його компонентів.

Отримані результати показали, що застосування концепції ООП із використанням класів, транзакцій, віртуального інтерфейсу, дозволяє ефективно організовувати структуру ТС за рахунок модульного підходу та ізольованості компонентів. Дозволяє легко розширити функціональність ТБ за рахунок наслідування класів та введення нових функцій, без порушення базової структури ТБ, що суттєво підвищує ефективність і гнучкість в процесі верифікації HDL-проектів.

Наступним кроком дослідження буде створення набору параметризованих класів для генерування версифікаційних компонентів ТС (драйвер, монітор, контрольна панель та ін.), що дозволить спростити процес верифікації IP-блоків контролерів послідовних інтерфейсів, таких як I2C, SPI, QSPI, LIN, CAN.

References

1. Roger Woods FPGA-based Implementation of Signal Processing Systems. 2nd ed. / Roger Woods, John McAllister, Gaye Lightbody, Ying Yi // Wiley, 2017. - 360p.
2. Mounir Maaref Architecting and Building High-Speed SoCs: Design, develop, and debug complex FPGA-based systems-on-chip // Packt Publishing, 2022. - 426 p.
3. Ross K. Snider Advanced Digital System Design using SoC FPGAs: An Integrated Hardware/Software Approach // Springer, 2023. – 455 p.
4. Frank Bruno The FPGA Programming Handbook: An essential guide to FPGA design for transforming ideas into hardware using SystemVerilog and VHDL. 2nd ed. / Frank Bruno, Guy Eschemann // Packt Publishing, 2024. - 550 p.
5. Vaibbhav Taraate Advanced HDL Synthesis and SOC Prototyping RTL Design Using Verilog // Springer, 2019. – 307 p. - https://doi.org/10.1007/978-981-10-8776-9
6. Janick Bergeron Writing Testbenches. Functional Verification of HDL Models // Kluwer Academic Publishers, 2000. – 354 p.
7. Chris Spear SystemVerilog for Verification. A Guide to Learning the Testbench Language Features. 3rd ed. / Chris Spear, Greg Tumbush // Springer, 2012. – 464 p.
8. Douglas J. Smith HDL Chip Design. A Practical Guide for Designing, Synthesizing and Simulating ASICs & FPGAs Using VHDL or Verilog // Doone Publications, 1997. – 446 p.
9. IEEE Std. 1364-2005 IEEE Standard for Verilog // IEEE Computer Society, 2006. – 560 p.
10. IEEE Std. 1076-2019 IEEE Standard for VHDL // IEEE Computer Society, 2019. – 671 p.
11. IEEE Std 1800™-2017 IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language // IEEE Computer Society, 2017. – 1314 p.
12. Janick Bergeron Writing Testbenches Using SystemVerilog // Springer, 2006. – 411 p.
13. Stuart Sutherland RTL Modeling with SystemVerilog for Simulation and Synthesis // Sutherland HDL Inc., 2017.- 456 p.
14. Stuart Sutherland SystemVerilog for Design. A Guide to Using SystemVerilog for Hardware Design and Modeling. 2nd ed. / Stuart Sutherland, Simon Davidmann, Peter Flake // Springer, 2006. – 418 p.
15. Donald Thomas Logic Design and Verification Using System Verilog // CreateSpace, 2016. – 311 p.
16. Pong P. Chu FPGA Prototyping by SystemVerilog Examples: Xilinx MicroBlaze MCS SoC. 2nd ed. // Wiley, 2018. - 656 p.
17. Vitis High-Level Synthesis. User Guide. UG1399 (v2022.2) // AMD (Xilinx), 2022. – 724 p. URL: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf
18. Tony Gaddis Starting Out with C++ from Control Structures to Objects. 9th ed. // Pearson, 2017. – 1344 p.