

**BỘ GIÁO DỤC VÀ ĐÀO TẠO  
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ KỸ THUẬT TP. HCM**

**KHOA ĐIỆN – ĐIỆN TỬ  
BỘ MÔN KỸ THUẬT MÁY TÍNH – VIỄN THÔNG**



**ĐỒ ÁN 2**

**THIẾT KẾ VÀ XÁC MINH  
CHỨC NĂNG IP UART DÙNG UVM**

**GVHD: ThS. Đậu Trọng Hiển**

**SVTH : Phạm Quang Hợp**

**MSSV: 22119178**

**TP. Hồ Chí Minh, tháng 1 năm 2026**

# Mục lục

<b>CHƯƠNG 1: GIỚI THIỆU .....</b>	<b>1</b>
1.1. Đặt vấn đề.....	1
1.2. Mục tiêu đề tài.....	2
1.3. Phương pháp nghiên cứu.....	2
1.4. Bố cục của đề tài .....	3
<b>CHƯƠNG 2: CƠ SỞ LÝ THUYẾT.....</b>	<b>4</b>
<b>2.1. Giới thiệu về UART (Universal Asynchronous Receiver/Transmitter).....</b>	<b>4</b>
2.1.1. Sơ lược về giao thức UART .....	4
2.1.2. Khung truyền UART .....	5
2.1.3. Các khối chức năng cơ bản của UART .....	5
2.1.4. Ý nghĩa tín hiệu và trạng thái thường gặp .....	6
<b>2.2. Giới thiệu phương thức xác minh UVM .....</b>	<b>7</b>
<b>2.3. Cấu trúc môi trường UVM.....</b>	<b>8</b>
2.3.1. UVM Testbench Top .....	8
2.3.2. UVM Test .....	9
2.3.3. UVM Environment .....	11
2.3.4. UVM Driver.....	12
2.3.5. UVM Sequencer .....	12
2.3.6. UVM Sequence.....	13
2.3.7. UVM Monitor .....	14
2.3.8. UVM Agent .....	15
2.3.9. UVM Scoreboard.....	17
<b>2.4. Cơ chế UVM .....</b>	<b>18</b>
2.4.1. Cơ chế pha .....	18
2.4.2. Cơ chế Factory .....	20
2.4.3. Cơ chế Config_db .....	20

## MỤC LỤC

---

2.4.4. Chức năng Coverage.....	21
<b>CHƯƠNG 3: XÂY DỰNG MÔI TRƯỜNG UVM KIỂM TRA THIẾT KẾ IP UART .....</b>	<b>22</b>
<b>3.1. YÊU CẦU CỦA THIẾT KẾ.....</b>	<b>22</b>
3.1.1. Yêu cầu của thiết kế UART .....	22
3.1.2. Yêu cầu của thiết kế môi trường UVM .....	22
<b>3.2. THIẾT KẾ IP UART .....</b>	<b>23</b>
3.2.1. Tổng quan về thiết kế .....	23
3.2.2. Sơ đồ RTL của thiết kế IP UART .....	26
3.2.3. Khối Baud_Rate_Gen .....	27
3.2.4. Khối FIFO (bao gồm cả FIFO_RX và TX).....	27
3.2.5. Khối UART_TX (UART Transmitter) .....	29
3.2.6. Khối UART_RX (UART Receiver) .....	30
<b>3.3. THIẾT KẾ MÔI TRƯỜNG UVM .....</b>	<b>32</b>
<b>3.4. XÂY DỰNG KẾ HOẠCH KIỂM TRA.....</b>	<b>34</b>
3.4.1. Kiểm tra Reset hệ thống.....	34
3.4.2. Kiểm tra chức năng truyền – nhận cơ bản (Write → Read) .....	35
3.4.3. Kiểm tra trạng thái FIFO_TX đầy (tx_full) và hành vi ghi/đọc theo burst ....	35
3.4.4. Kiểm tra đường nhận RX và cơ chế báo lỗi (incorrect_send) .....	35
3.4.5. Kiểm tra ngẫu nhiên và stress (random/system-like) .....	36
<b>CHƯƠNG 4: KẾT QUẢ VÀ ĐÁNH GIÁ .....</b>	<b>37</b>
4.1. KẾT QUẢ MÔ PHỎNG CHỨC NĂNG TRUYỀN – NHẬN UART .....	37
4.2. MÔ PHỎNG FSM TX/RX VÀ HÀNH VI FIFO .....	39
4.3. DÒNG DỮ LIỆU TRONG THANH GHI DỊCH RX .....	42
4.4. KẾT QUẢ BÁO CÁO QUA QUAN SÁT LOG MÔI TRƯỜNG .....	44
4.5. KẾT QUẢ BÁO CÁO COVERAGE .....	47
<b>CHƯƠNG 5: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN.....</b>	<b>51</b>

## **MỤC LỤC**

---

5.1. KẾT LUẬN.....	51
5.2. HƯỚNG PHÁT TRIỂN.....	52
<b>TÀI LIỆU THAM KHẢO.....</b>	<b>53</b>
<b>PHỤ LỤC.....</b>	<b>54</b>

# Danh sách hình ảnh

<b>Hình 2.1:</b> Vị trí của UART IP trong hệ thống .....	4
<b>Hình 2.2:</b> Định dạng khung truyền UART .....	5
<b>Hình 2.3:</b> Sơ đồ khối UART (dạng tổng quát).....	6
<b>Hình 2.4:</b> Hệ thống phân cấp trong môi trường UVM. ....	8
<b>Hình 2.5:</b> Cấu trúc UVM Testbench .....	9
<b>Hình 2.6:</b> Sơ đồ phân cấp của lớp uvm_test .....	10
<b>Hình 2.7:</b> Cấu trúc UVM Test .....	10
<b>Hình 2.8:</b> Sơ đồ phân cấp của lớp uvm_env .....	11
<b>Hình 2.9:</b> Cấu trúc UVM Environment .....	11
<b>Hình 2.10:</b> Sơ đồ phân cấp của lớp uvm_driver .....	12
<b>Hình 2.11:</b> Cấu trúc UVM Sequencer.....	13
<b>Hình 2.12:</b> Sơ đồ phân cấp của lớp uvm_sequencer.....	13
<b>Hình 2.13:</b> Cấu trúc UVM Sequence .....	14
<b>Hình 2.14:</b> Sơ đồ phân cấp của lớp uvm_sequence .....	14
<b>Hình 2.15:</b> Sơ đồ phân cấp của lớp uvm_monitor.....	15
<b>Hình 2.16:</b> Cấu trúc UVM Agent .....	16
<b>Hình 2.17:</b> Sơ đồ phân cấp của lớp uvm_agent.....	16
<b>Hình 2.18:</b> Cấu trúc Active Agent.....	16
<b>Hình 2.19:</b> Cấu trúc Passive Agent .....	16
<b>Hình 2.20.</b> Cấu trúc UVM Scoreboard .....	17
<b>Hình 2.21.</b> Sơ đồ phân cấp lớp của uvm_scoreboard .....	18
<b>Hình 2.22.</b> Cơ chế pha .....	19
<b>Hình 3.1:</b> Sơ đồ khối của thiết kế IP UART .....	23
<b>Hình 3.2:</b> Sơ đồ RTL của thiết kế UART trên QuestaSim .....	26
<b>Hình 3.3:</b> Sơ đồ RTL của khối Baud_Gen .....	27

## DANH SÁCH HÌNH ẢNH

---

<b>Hình 3.4:</b> Sơ đồ RTL rút gọn của khối FIFO .....	27
<b>Hình 3.5:</b> Sơ đồ RTL rút gọn của khối UART_TX.....	29
<b>Hình 3.6.</b> Sơ đồ FSM trong thiết kế UART_TX .....	29
<b>Hình 3.7.</b> Sơ đồ RTL rút gọn của khối UART_RX .....	30
<b>Hình 3.8.</b> Sơ đồ FSM trong thiết kế UART_RX .....	30
<b>Hình 3.9.</b> Sơ đồ khối xây dựng môi trường UVM kiểm tra thiết kế UART .....	32
<b>Hình 3.10.</b> UVM topology của môi trường kiểm thử UART .....	33
<b>Hình 4.1.</b> Kết quả mô phỏng chức năng truyền–nhận UART. ....	38
<b>Hình 4.2.</b> Kết quả mô phỏng FSM của TX/RX và cơ chế đẩy/đọc dữ liệu qua FIFO_RX.	
.....	40
<b>Hình 4.3.</b> Kết quả mô phỏng “Data Flow” .....	43
<b>Hình 4.4.</b> Kết quả báo cáo chạy mô phỏng qua màn hình LOG trên QuestaSim .....	45
<b>Hình 4.5.</b> Kết quả UVM report summary .....	46
<b>Hình 4.6.</b> Tóm tắt kết quả báo cáo coverage cho verify plan đã lên .....	47

# Danh sách bảng biểu

<b>Bảng 3.1:</b> Mô tả các chân của thiết kế IP UART .....	23
<b>Bảng 3.2:</b> Mô tả chức năng hoạt động của các tín hiệu điều khiển chính.....	26
<b>Bảng 4.1:</b> Kết quả báo cáo coverage cho covergroup UART_cg.....	47

# CHƯƠNG 1: GIỚI THIỆU

## 1.1. Đặt vấn đề

Trong những năm gần đây, cùng với sự phát triển mạnh mẽ của ngành công nghiệp vi mạch và xu hướng tích hợp hệ thống ngày càng phức tạp (SoC/ASIC/FPGA), yêu cầu về xác minh chức năng ngày càng trở nên khắt khe hơn. Các phương pháp kiểm tra truyền thống (viết test thủ công, kiểm tra theo kịch bản cố định) dần bộc lộ hạn chế như khó mở rộng, khó tái sử dụng và khó bao phủ hết các trường hợp biên trong thiết kế. Vì vậy, các phương pháp xác minh hiện đại theo hướng tự động hóa, tái sử dụng và đo lường mức độ bao phủ trở thành lựa chọn tất yếu trong quy trình phát triển chip.

UVM (Universal Verification Methodology – Phương pháp xác minh phổ quát) là một chuẩn xác minh dựa trên SystemVerilog, giúp xây dựng testbench theo kiến trúc phân lớp, hỗ trợ constrained-random, functional coverage, scoreboard, và cơ chế cấu hình/mở rộng linh hoạt. Nhờ khả năng chuẩn hóa luồng xác minh và tăng tính tái sử dụng, UVM đã và đang trở thành một trong những phương pháp hiệu quả nhất để nâng cao chất lượng xác minh, giảm thời gian debug, đồng thời cải thiện độ tin cậy khi triển khai các khối IP trong nhiều dự án khác nhau.

Trong lĩnh vực thiết kế số, UART (Universal Asynchronous Receiver/Transmitter) là một giao thức truyền thông nối tiếp bất đồng bộ cực kỳ phổ biến, xuất hiện rộng rãi trong các hệ thống nhúng, module giao tiếp ngoại vi, luồng debug/bring-up, và các kênh cấu hình điều khiển trên SoC. Một khía UART thực tế thường đi kèm nhiều thành phần quan trọng như bộ tạo baud rate, register file cấu hình, FIFO đệm dữ liệu, tín hiệu ngắn, và giao tiếp bus điều khiển (ví dụ: Wishbone). Do đó, việc xác minh UART không chỉ dừng ở kiểm tra truyền/nhận đúng dữ liệu, mà còn phải đảm bảo tính đúng đắn của cấu hình, điều kiện tràn/thiểu FIFO, xử lý reset, và các tình huống biên về timing/handshake.

Xuất phát từ nhu cầu đó, dự án tập trung xây dựng UART Verification IP (VIP) dựa trên UVM để xác minh chức năng cho một UART core có đầy đủ các khía TX/RX, baud rate generator, FIFO và hệ thanh ghi điều khiển. Môi trường UVM được tổ chức theo mô hình chuẩn với agent/driver/monitor/sequencer/sequence, kết hợp scoreboard để đối chiếu

dữ liệu mong đợi và coverage collector để đánh giá mức độ bao phủ chức năng. Thông qua dự án, mục tiêu không chỉ là kiểm chứng thiết kế UART đúng theo yêu cầu, mà còn hướng đến tạo ra một môi trường xác minh có tính tái sử dụng cao, dễ mở rộng testcase và phù hợp để áp dụng vào các hệ thống SoC có giao tiếp UART trong thực tế.

### 1.2. Mục tiêu đề tài

Mục tiêu chính của đề tài là xây dựng môi trường UVM hoàn chỉnh để xác minh các chức năng cơ bản của UART core (TX/RX, cấu hình, FIFO và điều khiển liên quan). Thiết kế RTL UART là nền tảng phục vụ cho mô phỏng và kiểm thử. Các mục tiêu cụ thể gồm:

- Thiết kế UART RTL gồm UART\_TX, UART\_RX, baud rate generator, register file và FIFO hoạt động đúng theo yêu cầu.
- Xây dựng môi trường UVM chuẩn (agent/driver/monitor/sequencer/sequence) để tạo stimulus và quan sát đầy đủ các giao tiếp của UART.
- Thiết kế scoreboard + functional coverage và phát triển các testcase nhằm kiểm tra toàn bộ chức năng cơ bản, quan sát kết quả qua waveform/log.
- Công cụ QuestaSim được sử dụng để biên dịch và chạy mô phỏng môi trường UVM..

### 1.3. Phương pháp nghiên cứu

- Phương pháp nghiên cứu tài liệu: Tìm kiếm và tổng hợp tài liệu liên quan đến UART và UVM từ sách chuyên ngành, tài liệu chuẩn SystemVerilog/UVM, tài liệu kỹ thuật về giao thức UART và các nguồn tham khảo uy tín phục vụ xây dựng môi trường xác minh.
- Phương pháp mô phỏng và đánh giá: Sử dụng QuestaSim để biên dịch và mô phỏng thiết kế RTL cùng testbench UVM; quan sát waveform, log mô phỏng, kiểm tra dữ liệu TX/RX, trạng thái FIFO và các thanh ghi điều khiển nhằm đánh giá tính đúng đắn của hệ thống.
- Phương pháp chọn lọc và tổng hợp kết quả: Tổng hợp kết quả mô phỏng, báo cáo các testcase tiêu biểu, mức độ coverage và các lỗi/điều chỉnh quan trọng; từ đó rút ra nhận xét và hoàn thiện báo cáo đề tài.

## 1.4. Bộ cục của đề tài

Đề tài gồm có 5 chương:

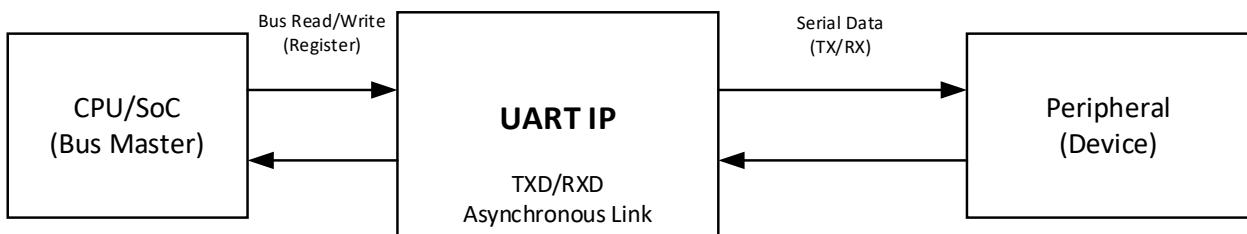
- Chương 1: Tổng quan – Trình bày bối cảnh, lý do chọn đề tài, mục tiêu, phương pháp thực hiện và bộ cục đồ án.
- Chương 2: Cơ sở lý thuyết – Giới thiệu kiến thức nền về giao thức UART và phương pháp xác minh UVM (kiến trúc testbench, sequence, driver/monitor, scoreboard, coverage).
- Chương 3: Xây dựng môi trường UVM kiểm tra thiết kế IP UART – Mô tả yêu cầu thiết kế, kiến trúc UART RTL (TX/RX, baud rate, FIFO, register file), đồng thời trình bày quá trình xây dựng môi trường UVM và cách xác minh các chức năng của UART.
- Chương 4: Kết quả và đánh giá – Trình bày kết quả mô phỏng, waveform/log, các testcase tiêu biểu và đánh giá coverage, kèm nhận xét.
- Chương 5: Kết luận và hướng phát triển – Tổng kết mức độ đáp ứng yêu cầu ban đầu, các điểm còn hạn chế và đề xuất hướng mở rộng (thêm testcase, mở rộng coverage, hỗ trợ thêm cấu hình/giao tiếp, nâng cấp VIP)..

# CHƯƠNG 2: CƠ SỞ LÝ THUYẾT

## 2.1. Giới thiệu về UART (Universal Asynchronous Receiver/Transmitter)

### 2.1.1. Sơ lược về giao thức UART

UART (Universal Asynchronous Receiver/Transmitter) là một chuẩn truyền thông nối tiếp bất đồng bộ dùng để truyền/nhận dữ liệu số giữa hai thiết bị. Khác với các giao thức đồng bộ (có đường clock riêng), UART không truyền clock, mà hai phía sẽ thỏa thuận trước các tham số truyền như baud rate, số bit dữ liệu, parity và stop bit. Nhờ cấu trúc đơn giản, UART được sử dụng rất phổ biến trong hệ thống nhúng, mạch giao tiếp ngoại vi, kenh debug/console, cũng như các khối IP giao tiếp trên SoC.



Hình 2.1. Vị trí của UART IP trong hệ thống

Trong một hệ thống số, UART thường đóng vai trò “cầu nối” giữa bus điều khiển của CPU và đường truyền nối tiếp. Dữ liệu từ CPU sẽ được UART đóng gói thành khung truyền nối tiếp để phát ra chân TX, đồng thời UART cũng có khả năng thu dữ liệu từ chân RX, giải mã khung truyền và đưa dữ liệu về cho CPU thông qua thanh ghi/FIFO. Vì UART liên quan trực tiếp đến luồng giao tiếp dữ liệu, việc hiểu rõ nguyên lý khung truyền, cơ chế đồng bộ theo baud rate và các cờ trạng thái lỗi là nền tảng quan trọng để xây dựng môi trường xác minh chức năng.

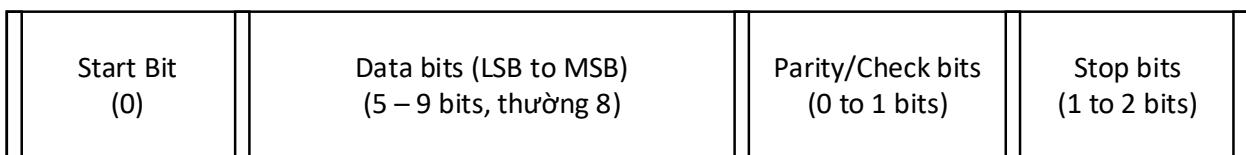
Quá trình truyền – nhận cơ bản của UART có thể mô tả như sau:

- CPU/thiết bị điều khiển cấu hình UART (baud rate, data bits, parity, stop bit...).
- Khi truyền, UART lấy dữ liệu song song từ bus/CPU, đóng gói thành khung UART và dịch bit ra đường TX theo tốc độ baud.
- Khi nhận, UART phát hiện start bit, lấy mẫu dữ liệu theo nhịp baud (thường có oversampling), khôi phục byte dữ liệu và đưa vào FIFO/ thanh ghi nhận.

- UART cập nhật các cờ trạng thái (TX empty, RX full, overrun, parity error, framing error...) và có thể phát ngắt để báo cho CPU xử lý.

### 2.1.2. Khung truyền UART

Một khung truyền UART điển hình gồm các thành phần: Start bit – Data bits – (Parity) – Stop bit(s). Đường truyền UART ở trạng thái rỗng thường ở mức logic ‘1’. Khi bắt đầu truyền, UART phát start bit = 0 để báo hiệu bắt đầu khung, sau đó phát các bit dữ liệu (thường LSB trước), có thể kèm parity để kiểm tra lỗi, và kết thúc bằng một hoặc nhiều stop bit = 1.



Hình 2.2. Định dạng khung truyền UART

Mô tả điển hình:

- Start bit: 1 bit ‘0’, dùng để đồng bộ thời điểm bắt đầu.
- Data bits: 5 đến 9 bit (phổ biến 8 bit), truyền LSB trước.
- Parity bit (tùy chọn): even/odd parity để phát hiện lỗi đơn giản.
- Stop bit(s): 1 hoặc 2 bit ‘1’, đánh dấu kết thúc khung và tạo khoảng nghỉ giữa các khung liên tiếp.

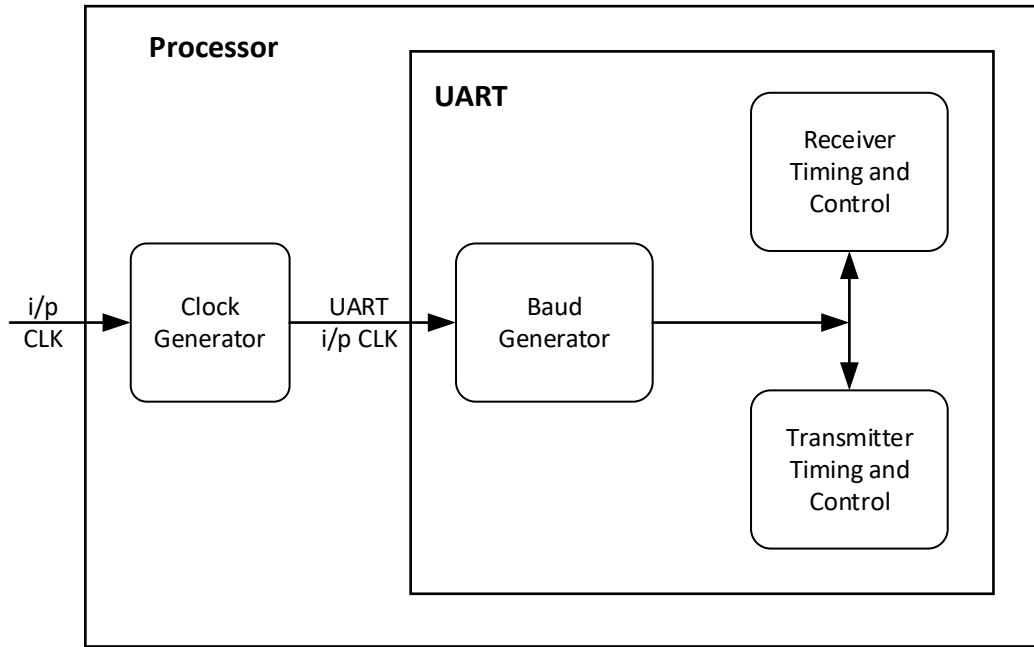
Ngoài ra, UART yêu cầu hai phía phải thống nhất baud rate (ví dụ 9600, 115200 bps...). Baud rate quyết định thời gian tồn tại của mỗi bit, và sai lệch baud giữa hai thiết bị nếu vượt ngưỡng cho phép có thể gây lỗi lấy mẫu (framing error/bit error).

### 2.1.3. Các khối chức năng cơ bản của UART

Các khối chức năng cơ bản thường gồm:

- Baud Rate Generator: chia clock hệ thống để tạo nhịp bit (có thể hỗ trợ oversampling).
- UART Transmitter (TX): thanh ghi dịch và logic đóng gói khung (start/data/parity/stop).

- UART Receiver (RX): phát hiện start bit, lấy mẫu dữ liệu theo nhịp baud, kiểm tra parity/stop.
- Register File / Control & Status: chứa thanh ghi cấu hình và cờ trạng thái/lỗi.



Hình 2.3. Sơ đồ khái niệmUART (dạng tổng quát)

Mặc dù cùng tuân theo nguyên lý khung truyền, thiết kế UART trong thực tế có thể mở rộng thêm FIFO, ngắn, nhiều chế độ cấu hình,...

- FIFO: đệm dữ liệu TX/RX để giảm nguy cơ mất dữ liệu khi CPU xử lý chậm.
- Interrupt Logic: phát ngắt khi RX có dữ liệu, TX rỗng, hoặc khi có lỗi đường truyền.

#### 2.1.4. Ý nghĩa tín hiệu và trạng thái thường gặp

Một UART tối thiểu thường có các tín hiệu giao tiếp nối tiếp:

- TXD (Transmit): ngõ ra truyền nối tiếp.
- RXD (Receive): ngõ vào nhận nối tiếp.

Ngoài ra, trong các UART IP tích hợp trên SoC thường có thêm:

- Interrupt (INT): báo sự kiện (RX ready, TX empty, lỗi...).
- Clock/Reset nội bộ: dùng cho logic đồng bộ bên trong UART (UART truyền bắt đồng bộ ra ngoài nhưng mạch vẫn chạy đồng bộ theo clock hệ thống).

- Giao tiếp bus (Wishbone/APB/AHB...): CPU ghi/đọc thanh ghi cấu hình và dữ liệu.

Các cờ trạng thái/lỗi quan trọng khi xác minh UART gồm:

- TX Empty / TX Ready: báo bộ phát rỗng hoặc sẵn sàng nhận dữ liệu mới.
- RX Full / RX Valid: báo bộ nhận có dữ liệu.
- Overrun Error: FIFO/ thanh ghi nhận đầy nhưng vẫn có byte mới đến.
- Parity Error: parity không khớp (khi bật parity).
- Framing Error: stop bit không đúng mức ‘1’ (thường do sai baud hoặc nhiễu).

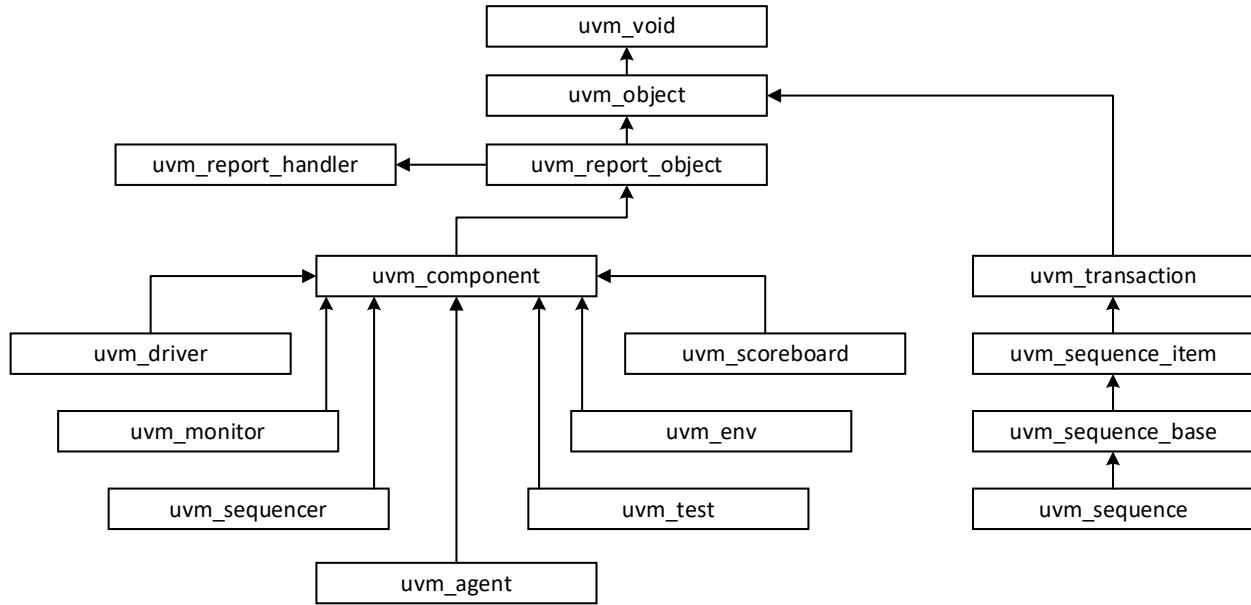
Do UART là truyền thông bất đồng bộ, dữ liệu ngõ ra sẽ phụ thuộc trực tiếp vào cấu hình baud và thời điểm lấy mẫu. Vì vậy, khi mô phỏng/xác minh cần đảm bảo mô hình truyền/nhận tuân theo đúng bit time, đúng thứ tự bit và đúng điều kiện kiểm tra parity/stop để đánh giá chính xác hành vi của thiết kế UART.

### 2.2. Giới thiệu phương thức xác minh UVM

UVM (Universal Verification Methodology) là một chuẩn phương pháp được dùng rộng rãi để xây dựng môi trường xác minh cho thiết kế số và SoC trong lĩnh vực vi mạch. UVM dựa trên SystemVerilog, tận dụng các nguyên lý của lập trình hướng đối tượng (OOP) nhằm tạo ra testbench có cấu trúc rõ ràng, dễ mở rộng và đặc biệt tái sử dụng cao. Đây là điểm mạnh nổi bật của UVM: thay vì viết lại testbench cho từng dự án, người thiết kế có thể tái dùng các khôi đã có như *agent*, *driver*, *monitor*, *scoreboard*... giúp tiết kiệm thời gian phát triển, đồng thời tăng tính nhất quán trong quy trình xác minh.

Bên cạnh đó, UVM cung cấp một hệ thống các lớp nền tảng (base classes) làm “khung xương” để người dùng phát triển các lớp nâng cao thông qua cơ chế kế thừa và bổ sung chức năng theo nhu cầu của từng môi trường xác minh. Cấu trúc này được thể hiện qua hệ thống phân cấp (như Hình 2.4).

Theo hệ thống phân cấp đó, lớp *uvm\_object* bắt nguồn từ *uvm\_void*. Trong đó, *uvm\_void* có thể xem là lớp gốc mang tính trừu tượng, hầu như không chứa dữ liệu hay phương thức triển khai cụ thể, đóng vai trò nền cho các đối tượng UVM được tạo ra. Trên nền này, *uvm\_object* cung cấp các khả năng thao tác và quản lý đối tượng như tạo đối tượng, sao chép, in thông tin... phục vụ cho cấu hình và điều khiển hoạt động của testbench.



Hình 2.4. Hệ thống phân cấp trong môi trường UVM.

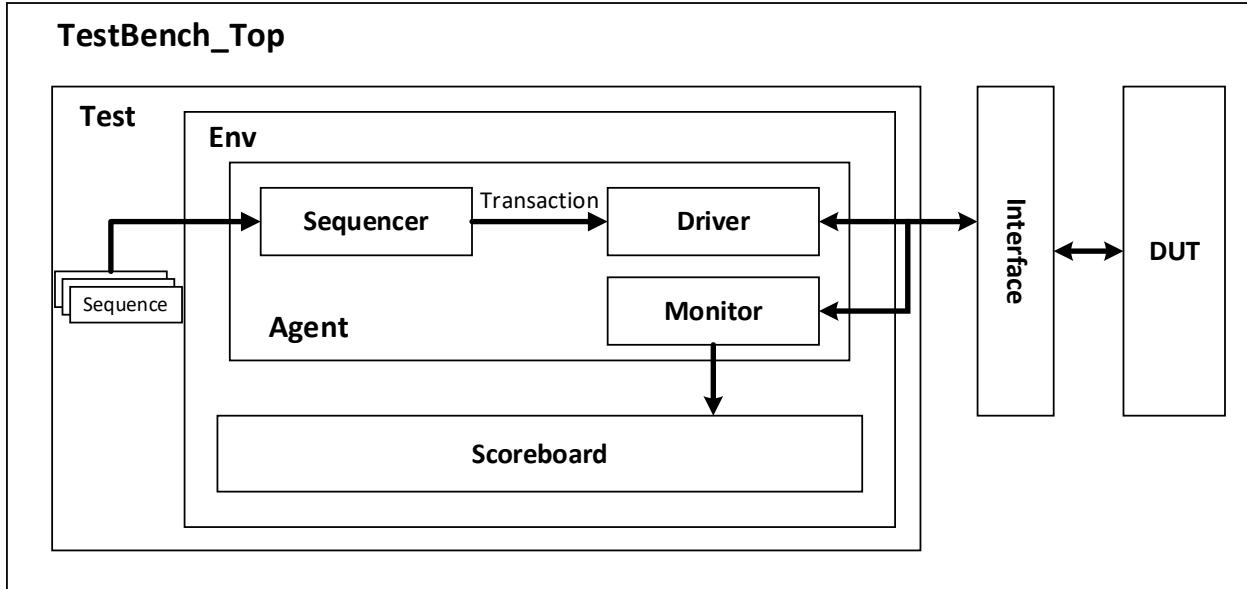
Ngoài ra, UVM còn có lớp *uvm\_report\_object* đảm nhiệm cơ chế báo cáo và ghi log. Hệ thống báo cáo chuẩn của UVM bao gồm bốn mức cơ bản: *info*, *warning*, *error* và *fatal*, giúp người dùng theo dõi tiến trình mô phỏng, phát hiện lỗi và dừng mô phỏng khi gặp lỗi nghiêm trọng.

Từ các lớp nền này, UVM tách ra hai nhánh chính thường gặp: nhánh *component* và nhánh *sequence*. Ở nhánh *uvm\_component*, các lớp dùng để mô tả những thành phần xác minh như *driver*, *monitor*, *agent*, *scoreboard*... và chúng hoạt động theo các phase trong mô phỏng. Trong khi đó, nhánh *uvm\_sequence* tập trung mô tả chuỗi kích thích (stimulus) và/hoặc đối tượng dữ liệu giao dịch (transaction) được tạo ra để các thành phần trong môi trường xác minh sử dụng, trao đổi và vận hành trong quá trình kiểm thử.

## 2.3. Cấu trúc môi trường UVM

### 2.3.1. UVM Testbench Top

Testbench Top (TB\_top) là mô-đun đóng vai trò điểm kết nối trung tâm giữa DUT và toàn bộ các thành phần của môi trường xác minh. Việc liên kết này thường được thực hiện thông qua một interface, như minh họa ở Hình 2.5.



Hình 2.5. Cấu trúc UVM Testbench

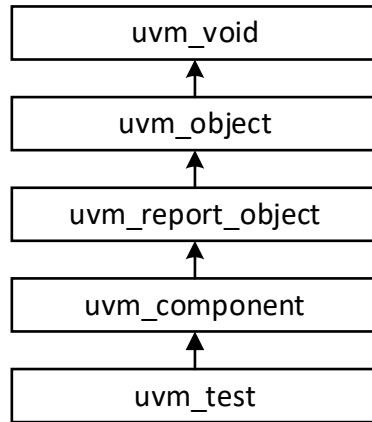
Một cấu trúc TB\_top điển hình thường gồm các thành phần sau:

- *DUT instance*: khối thiết kế cần được kiểm tra/xác minh.
- *Interface instance*: lớp giao tiếp trung gian giúp môi trường xác minh trao đổi tín hiệu với DUT.
- Phương thức *run\_test()*: hàm được gọi để khởi chạy các testcase trong testbench.
- Thiết lập *virtual interface* qua *config\_db*: cấu hình để môi trường UVM truy cập DUT thông qua *interface* ảo.
- *Clock* và *Reset*: tạo và điều khiển xung nhịp/reset phục vụ quá trình kiểm tra DUT.
- *Wave dump*: sinh file dạng sóng (waveform) nhằm quan sát biến thiên tín hiệu khi mô phỏng, hỗ trợ debug và phân tích lỗi.

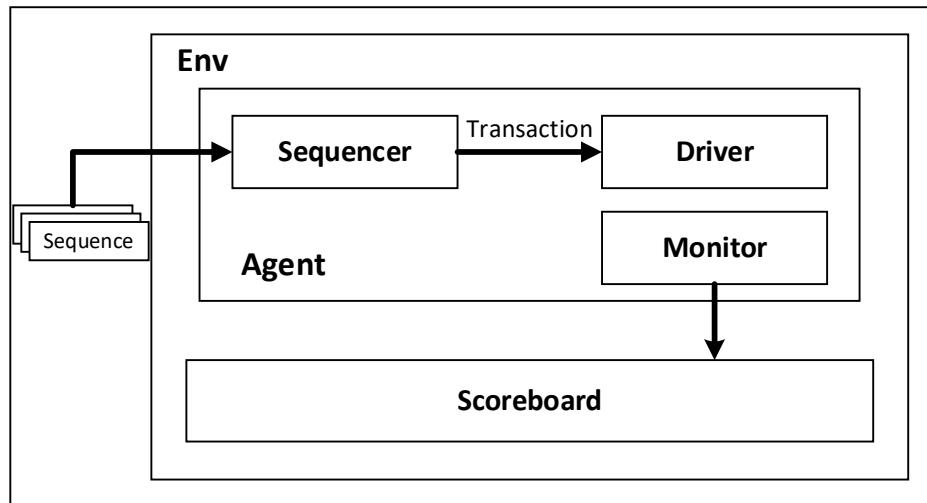
### 2.3.2. UVM Test

*UVM Test* có cấu trúc phân cấp như được minh họa ở *Hình 2.6*. Trong đó, lớp *uvm\_test* đại diện cho các trường hợp kiểm thử nhằm đánh giá chức năng và tính đúng đắn của *DUT*. Kế hoạch xác minh (verification plan) sẽ liệt kê đầy đủ các tính năng, hạng mục cần xác minh, đồng thời xác định các bài kiểm tra cần thiết để bao phủ từng tính năng đó. Nhờ cách tổ chức này, thay vì phải viết lại nhiều đoạn mã cho các testcase khác nhau, ta có thể tái sử

dụng cùng một môi trường xác minh, chỉ cần thay đổi cấu hình hoặc sequence tương ứng cho từng trường hợp kiểm tra.



Hình 2.6. Sơ đồ phân cấp của lớp `uvm_test`



Hình 2.7. Cấu trúc `UVM Test`

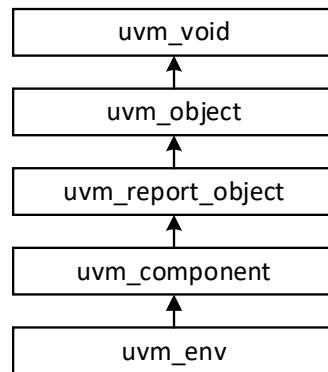
Các bước xây dựng một UVM Test thường gồm:

- Tạo lớp test do người dùng định nghĩa, kế thừa từ `uvm_test` và đăng ký với `factory`.
- Khai báo và khởi tạo `environment`, `sequence` và các đối tượng cấu hình theo yêu cầu kiểm thử.
- Kích hoạt và chạy `sequence` để bắt đầu quá trình kiểm tra.

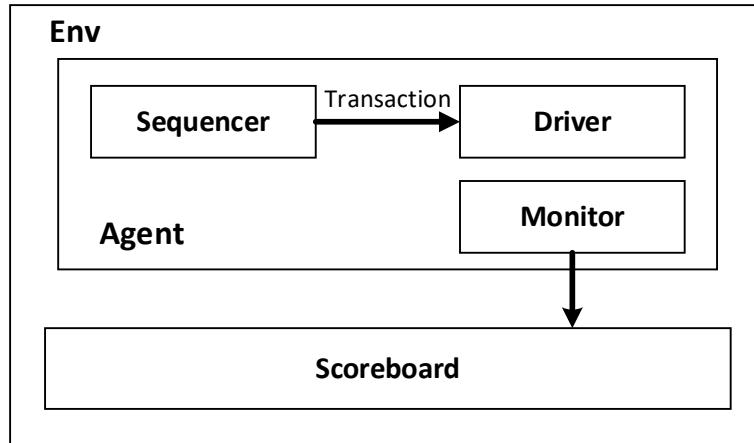
Thông thường, `UVM Test` sẽ được thực thi trong `Testbench Top` thông qua lời gọi hàm `run_test()`.

### 2.3.3. UVM Environment

UVM Environment là lớp `uvm_env` (như thể hiện ở Hình 2.8), đóng vai trò là “khung môi trường” theo dạng phân cấp, nơi tập hợp và quản lý các thành phần xác minh như agent, scoreboard, coverage và các khối liên quan khác (minh họa ở Hình 2.9). Lớp `uvm_env` cung cấp cơ chế để cấu hình, khởi tạo và điều phối hoạt động của các thành phần này nhằm thực hiện các tác vụ kiểm tra đúng theo yêu cầu đề ra.



Hình 2.8. Sơ đồ phân cấp của lớp `uvm_env`



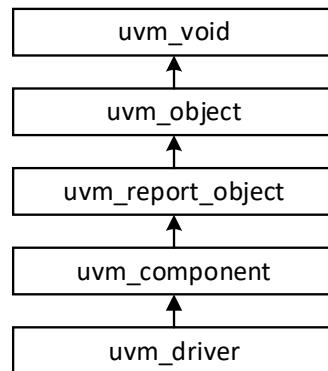
Hình 2.9. Cấu trúc UVM Environment

Các bước xây dựng UVM Environment thường gồm:

- Tạo lớp environment do người dùng định nghĩa, kế thừa từ `uvm_env` và đăng ký lên factory.
- Trong `build_phase`, khởi tạo agent, scoreboard và các thành phần xác minh khác.
- Thực hiện kết nối agent với scoreboard thông qua TLM port để truyền transaction và phục vụ so khớp kết quả.

### 2.3.4. UVM Driver

UVM Driver có cấu trúc phân cấp như minh họa ở Hình 2.10. Thành phần này đảm nhiệm việc điều khiển và phát tín hiệu từ môi trường xác minh xuống DUT thông qua các giao thức phần cứng như bus, truyền thông nối tiếp, hoặc bất kỳ giao thức nào mà DUT hỗ trợ. Nói cách khác, `uvm_driver` chịu trách nhiệm biến các transaction/sequence thành các tín hiệu chân thực trên interface, đồng thời đảm bảo việc phát dữ liệu và tín hiệu điều khiển diễn ra đúng trình tự và đúng thời điểm.



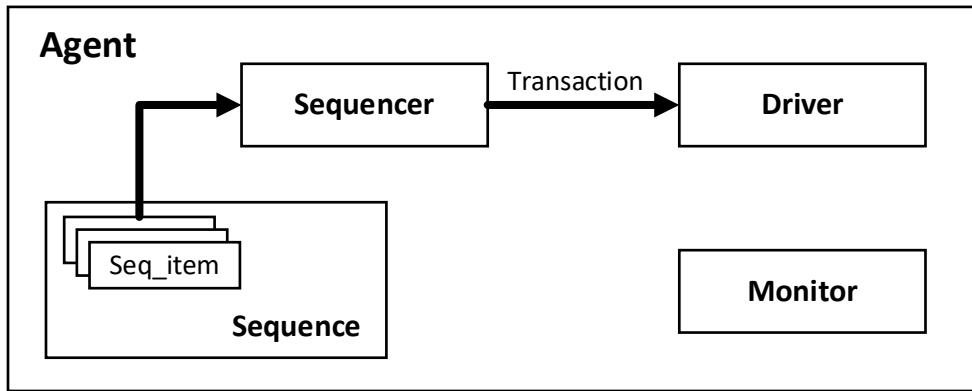
Hình 2.10. Sơ đồ phân cấp của lớp `uvm_driver`

Các bước tạo UVM Driver thường gồm:

- Tạo lớp *driver* do người dùng định nghĩa, kế thừa từ `uvm_driver` và đăng ký lên *factory*.
- Khai báo *virtual interface* để truy cập *interface thật* thông qua `config_db` trong *build\_phase*.
- Thực thi *build\_phase* và lấy *virtual interface* từ cơ sở dữ liệu cấu hình.
- Thực thi *run\_phase* để nhận các *item* từ *sequence* (qua *sequencer*) và drive xuống *DUT* bằng *virtual interface*.

### 2.3.5. UVM Sequencer

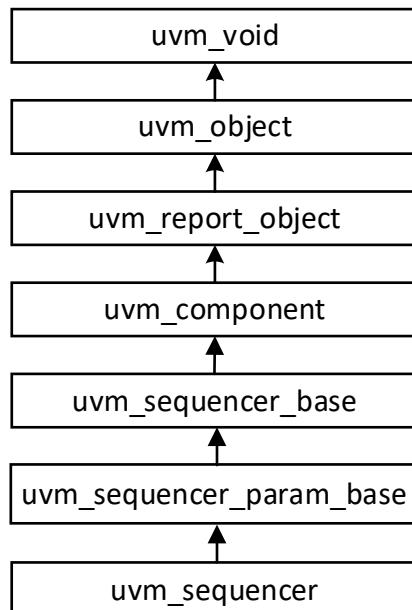
*UVM Sequencer* là thành phần trung gian dùng để thiết lập kết nối giữa `uvm_sequence` và `uvm_driver` như minh họa ở Hình 2.11. Nhiệm vụ chính của *sequencer* là chuyển các *sequence/sequence item* từ phía tạo kích thích (*sequence*) sang phía thực thi (*driver*); sau đó driver sẽ dựa trên các item này để phát tín hiệu xuống *DUT*.



Hình 2.11. Cấu trúc UVM Sequencer

Lớp uvm\_sequencer giữ vai trò quan trọng trong việc tạo và điều phối chuỗi kích thích, đồng thời kiểm soát luồng trao đổi sequence giữa các thành phần trong môi trường kiểm tra (đảm bảo thứ tự, cơ chế cấp phát item, arbitration khi có nhiều sequence...).

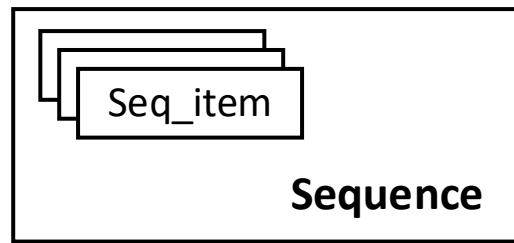
Sự phân cấp của lớp uvm\_sequencer được cấu trúc như sau:



Hình 2.12. Sơ đồ phân cấp của lớp uvm\_sequencer

### 2.3.6. UVM Sequence

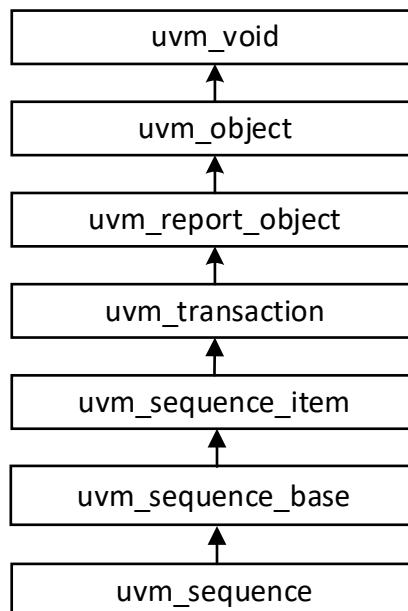
*UVM Sequence* là thành phần chứa các đơn vị dữ liệu giao dịch (*uvm\_sequence\_item*) như minh họa ở Hình 2.13. Các gói dữ liệu mà *sequence* tạo ra hoặc thu thập được sẽ được gửi đến *driver* thông qua *sequencer*, từ đó *driver* chuyển đổi chúng thành tín hiệu thực để tác động lên *DUT*.



Hình 2.13. Cấu trúc UVM Sequence

Lớp *uvm\_sequence* cung cấp nhiều phương thức và thuộc tính nhằm điều khiển luồng kích thích và theo dõi trạng thái dữ liệu trong quá trình kiểm tra. Nhờ đó, *sequence* có thể quản lý thứ tự, tần suất thực hiện các hoạt động kiểm thử, đồng thời xử lý các tình huống/trạng thái khác nhau của chuỗi (ví dụ: lặp, dừng, chuyển pha, hoặc ràng buộc dữ liệu).

Sự phân cấp của lớp *uvm\_sequence* được cấu trúc như sau:

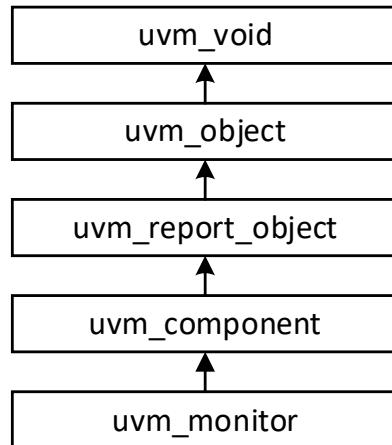


Hình 2.14. Sơ đồ phân cấp của lớp *uvm\_sequence*

### 2.3.7. UVM Monitor

*UVM Monitor* được dùng để quan sát và thu thập dữ liệu đầu ra/hoạt động của *DUT* thông qua *virtual interface*, sau đó chuyển đổi các tín hiệu đã lấy mẫu thành dạng *transaction/sequence item* như minh họa ở Hình 2.5. Những *transaction* này sẽ được phát

(publish) sang các khối khác trong môi trường xác minh như *uvm\_scoreboard*, *coverage collector*,... để phục vụ đổi chiểu kết quả và thống kê độ bao phủ.



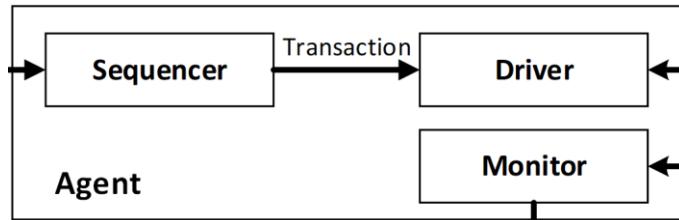
Hình 2.15. Sơ đồ phân cấp của lớp `uvm_monitor`

Các bước xây dựng UVM Monitor thường gồm:

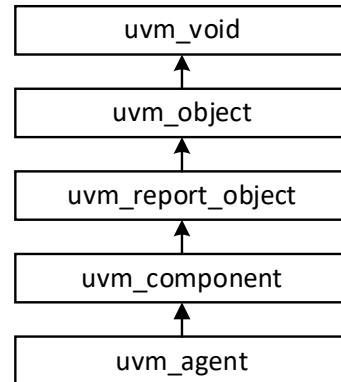
- Tạo lớp monitor do người dùng định nghĩa, kế thừa từ `uvm_monitor` và đăng ký lên factory.
- Khai báo virtual interface để truy cập interface thật thông qua `config_db` trong `build_phase`.
- Khai báo cổng giao tiếp (analysis port) để phát các transaction ra ngoài.
- Triển khai `build_phase` để lấy virtual interface từ cơ sở dữ liệu cấu hình.
- Triển khai `run_phase` để lấy mẫu tín hiệu từ DUT bằng virtual interface, đóng gói thành transaction; sau đó dùng `write()` (hoặc analysis port) để gửi transaction sang scoreboard/coverage.

### 2.3.8. UVM Agent

*UVM Agent* là thành phần có nhiệm vụ tập hợp và liên kết các khối *driver*, *monitor* và *sequencer* như minh họa ở Hình 2.16. Lớp `uvm_agent` cũng có hệ thống phân cấp lớp riêng như thể hiện ở Hình 2.17. *Agent* giúp đóng gói các thành phần liên quan đến một giao tiếp/protocol thành một khối thống nhất, từ đó dễ cấu hình, tái sử dụng và mở rộng môi trường xác minh.



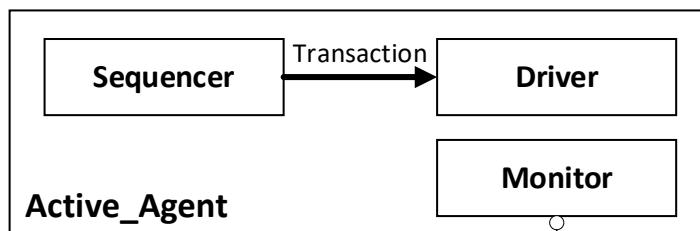
Hình 2.16. Cấu trúc UVM Agent



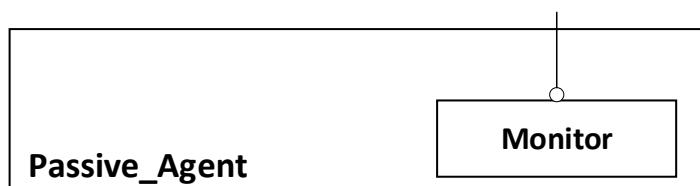
Hình 2.17. Sơ đồ phân cấp của lớp `uvm_agent`

Các bước xây dựng *UVM Agent* thường gồm:

- Tạo lớp *agent* do người dùng định nghĩa, kế thừa `uvm_agent` và đăng ký lên *factory*.
- Trong *build\_phase*, khởi tạo *driver*, *monitor* và *sequencer* nếu *agent* ở chế độ *active*; nếu là *passive* thì chỉ khởi tạo *monitor*.
- Trong *connect\_phase*, thực hiện kết nối giữa *driver* và *sequencer* để *driver* nhận *sequence item* từ *sequencer*.



Hình 2.18. Cấu trúc Active Agent

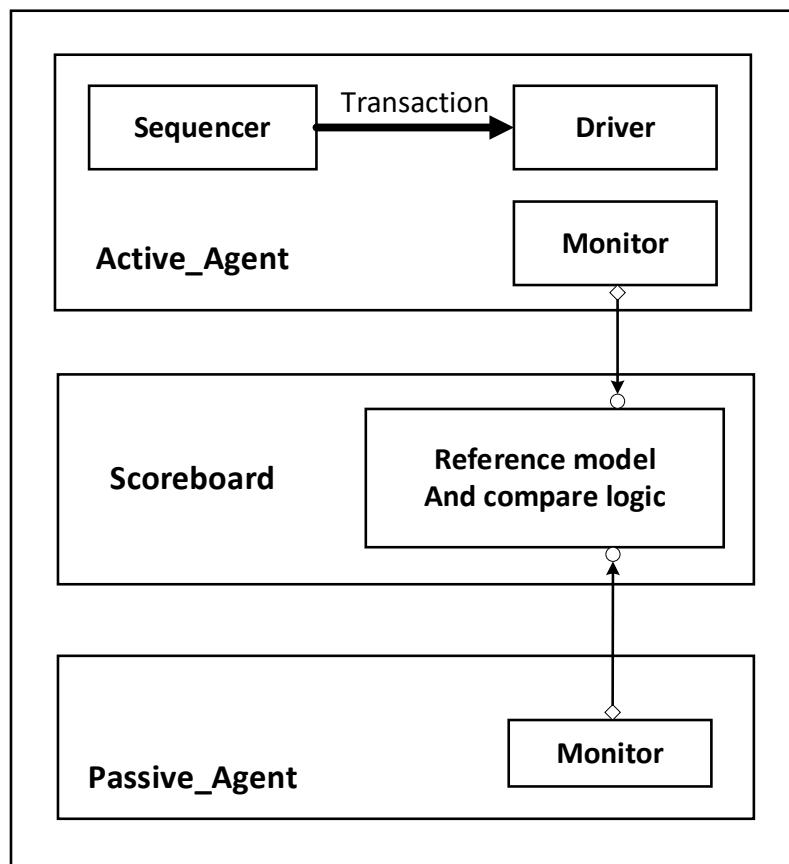


Hình 2.19. Cấu trúc Passive Agent

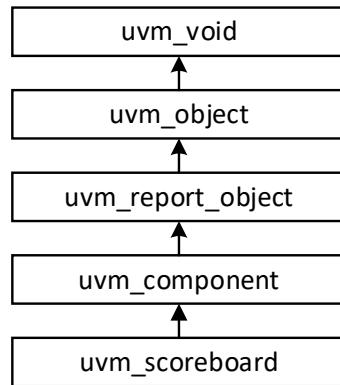
Về chức năng, *active agent* có nhiệm vụ tạo kích thích và drive dữ liệu xuống *DUT*. Như *Hình 2.18*, khi *agent* hoạt động ở chế độ active thì cả ba thành phần *driver-monitor-sequencer* sẽ được khởi tạo đầy đủ. Ngược lại, *passive agent* không tạo stimulus cho *DUT* mà chỉ dùng để quan sát và thu thập dữ liệu, thường phục vụ mục đích *kiểm tra/coverage*; vì vậy như *Hình 2.19*, *passive agent* chỉ khởi tạo *monitor*.

### 2.3.9. UVM Scoreboard

*UVM Scoreboard* là thành phần dùng để kiểm tra và đánh giá chức năng của *DUT*. *uvm\_scoreboard* sẽ nhận các *transaction/sequence item* từ *monitor* thông qua cơ chế analysis (công phân tích) để phục vụ việc so sánh kết quả. Sau khi nhận được các đối tượng dữ liệu, *scoreboard* có thể tự thực hiện tính toán để suy ra giá trị mong đợi từ dữ liệu đầu vào, hoặc chuyển dữ liệu này sang *reference model* nhằm tạo ra kết quả dự đoán. *Reference model* hoạt động như một “mô hình chuẩn”, mô phỏng hành vi đúng của thiết kế để làm cơ sở đối chiếu. *Hình 2.20* minh họa cấu trúc của *uvm\_scoreboard*.



Hình 2.20. Cấu trúc UVM Scoreboard



Hình 2.21. Sơ đồ phân cấp lớp của `uvm_scoreboard`

Các bước xây dựng *UVM Scoreboard* thường gồm:

- Tạo lớp *scoreboard* do người dùng định nghĩa, kế thừa từ `uvm_scoreboard` và đăng ký lên *factory*.
- Triển khai `build_phase` và tạo các cổng giao tiếp TLM/analysis cần thiết.
- Ghi đè (override) phương thức `write()` để nhận *transaction* từ *monitor*.
- Triển khai `run_phase` để thực hiện kiểm tra/so khớp kết quả của *DUT* xuyên suốt thời gian mô phỏng.

## 2.4. Cơ chế UVM

### 2.4.1. Cơ chế pha

Trong UVM, phase (pha) được hiểu như một cơ chế/chuỗi phương thức giúp đồng bộ và điều phối hoạt động của toàn bộ môi trường xác minh. Khi mô phỏng bắt đầu, UVM sẽ lần lượt chạy các phase theo trật tự phân cấp, nhờ đó các thành phần trong testbench (*test*, *env*, *agent*, *driver*, *monitor*, *scoreboard*, ...) được thực thi đúng tiến trình và thống nhất với nhau.

build	
connect	
end_of_elaberation	
start_of_elaberation	
run	pre_reset
	reset
	reset
	pre_configure
	configure
	post_configure
	pre_main
	main
	post_main
	pre_shutdown
	shutdown
	post_shutdown
extract	
check	
report	
final	

Hình 2.22. Cơ chế pha

Hình 2.22 minh họa hệ thống phase trong UVM, các phase được kích hoạt theo hướng từ trên xuống dưới, tạo ra sự đồng bộ giữa tất cả các khối trong môi trường UVM. Về nguyên lý, quá trình này thường được mô tả theo ba giai đoạn chính:

- **Pha khởi tạo:** Diễn ra ở đầu mô phỏng, bao gồm các phase từ build đến *start\_of\_elaboration*. Tại giai đoạn này, môi trường UVM thực hiện việc xây dựng cấu trúc, thiết lập cấu hình và kết nối các thành phần theo hệ thống phân cấp. Do các

phase khởi tạo chủ yếu được triển khai dưới dạng function, nên chúng không tiêu tốn thời gian mô phỏng.

- **Pha thực thi:** Đây là giai đoạn tiến hành đưa stimulus/dữ liệu kiểm thử vào DUT và quan sát phản hồi. Theo Hình 2.22, các phase từ *pre\_reset* đến *post\_shutdown* có thể được tổ chức để chạy song song giữa các thành phần. Vì nhóm phase này thường được triển khai dưới dạng task, nên chúng có tiêu tốn thời gian mô phỏng, và các khói trong môi trường UVM cũng hoạt động đồng thời trong giai đoạn này.
- **Pha kết thúc:** Sau khi mô phỏng hoàn tất, UVM chuyển sang giai đoạn thu thập và tổng hợp. Ở pha này, hệ thống sẽ gom kết quả kiểm thử, tạo báo cáo từ *scoreboard* và *functional coverage*, từ đó đánh giá các trường hợp đã được kiểm tra, mức độ *coverage* đạt được, cũng như xác định các lỗi hoặc mục tiêu chưa đáp ứng yêu cầu.

#### 2.4.2. Cơ chế Factory

*UVM Factory* là một cơ chế giúp tăng tính linh hoạt và khả năng mở rộng của môi trường xác minh. Cụ thể, Factory cho phép thay thế (override) các lớp đối tượng đã có bằng các lớp kế thừa (derived classes) mà không cần sửa trực tiếp mã nguồn của môi trường. UVM cung cấp các macro để đăng ký lớp với *factory*, đồng thời cung cấp các phương thức cho phép thực hiện cơ chế ghi đè lớp cơ sở bằng lớp dẫn xuất khi tạo đối tượng.

Trong thực tế, *Factory* được dùng để đăng ký các thành phần trong môi trường UVM, nhờ đó người dùng có thể dễ dàng thay thế hoặc cấu hình lại các thành phần này theo từng testcase. Hai macro thường dùng để đăng ký đối tượng lên *Factory* là *uvm\_component\_utils* và *uvm\_object\_utils*. Nhờ cơ chế *Factory*, môi trường UVM trở nên linh hoạt hơn trong quá trình xây dựng, cấu hình và đặc biệt nâng cao khả năng tái sử dụng giữa nhiều bài kiểm thử và nhiều dự án khác nhau.

#### 2.4.3. Cơ chế Config\_db

Một testbench khi được cấu hình theo cơ chế truy xuất từ cơ sở dữ liệu cấu hình (*config\_db*) sẽ trở nên linh hoạt hơn, vì cho phép thiết lập nhiều cấu hình khác nhau cho các thành phần trong môi trường xác minh UVM. Cơ chế *config\_db* hỗ trợ hai phương thức chính là *set()* và *get()*. Trong đó, *set()* dùng để đưa/lưu đối tượng cấu hình (config object)

vào cơ sở dữ liệu, còn `get()` dùng để lấy/nhận lại đối tượng cấu hình đó để các thành phần UVM sử dụng trong quá trình hoạt động.

### 2.4.4. Chức năng Coverage

Xác minh thiết kế là quá trình kiểm tra toàn bộ chức năng của thiết kế, vì vậy *coverage* được sử dụng để theo dõi và đánh giá mức độ mà các chức năng đó đã được kích hoạt trong quá trình kiểm thử. *Coverage* giúp xác định phần nào đã được kiểm tra đầy đủ, phần nào còn thiếu để bổ sung testcase phù hợp. Có hai loại *coverage* thường dùng:

- *Code Coverage*: Dùng để đo lường mức độ các thành phần trong mã RTL đã được thực thi khi mô phỏng, như *statement/branch/condition/FSM*.... Những đoạn mã chưa được “chạy qua” sẽ được xem xét lại để bổ sung stimulus, đảm bảo thiết kế được kiểm tra ở nhiều trường hợp khác nhau.
- *Functional Coverage*: Dùng để đo mức độ các tính năng/ý đồ chức năng đã được kiểm tra, thường được định nghĩa bằng *covergroup/coverpoint/cross*. Các mục tiêu chức năng chưa đạt sẽ được bổ sung bằng các bộ dữ liệu đầu vào và kịch bản kiểm thử tương ứng, nhằm đảm bảo mọi chức năng quan trọng của thiết kế đều được xác minh và hoạt động đúng yêu cầu.

# CHƯƠNG 3: XÂY DỰNG MÔI TRƯỜNG UVM

## KIỂM TRA THIẾT KẾ IP UART

### 3.1. YÊU CẦU CỦA THIẾT KẾ

#### 3.1.1. Yêu cầu của thiết kế UART

Thiết kế UART gồm các khối chức năng chính:

- Khối truyền (UART\_TX): thực hiện đóng gói dữ liệu, phát start bit, data bit, parity và stop bit qua đường TX.
- Khối nhận (UART\_RX): lấy mẫu tín hiệu RX, kiểm tra parity/stop bit và giải mã dữ liệu nhận.
- Baud Rate Generator: tạo xung baud phục vụ truyền và nhận dữ liệu.
- FIFO và Register File: lưu trữ tạm thời dữ liệu truyền/nhận, đồng thời cung cấp tín hiệu trạng thái như tx\_full, rx\_empty, tx\_done, rx\_done.

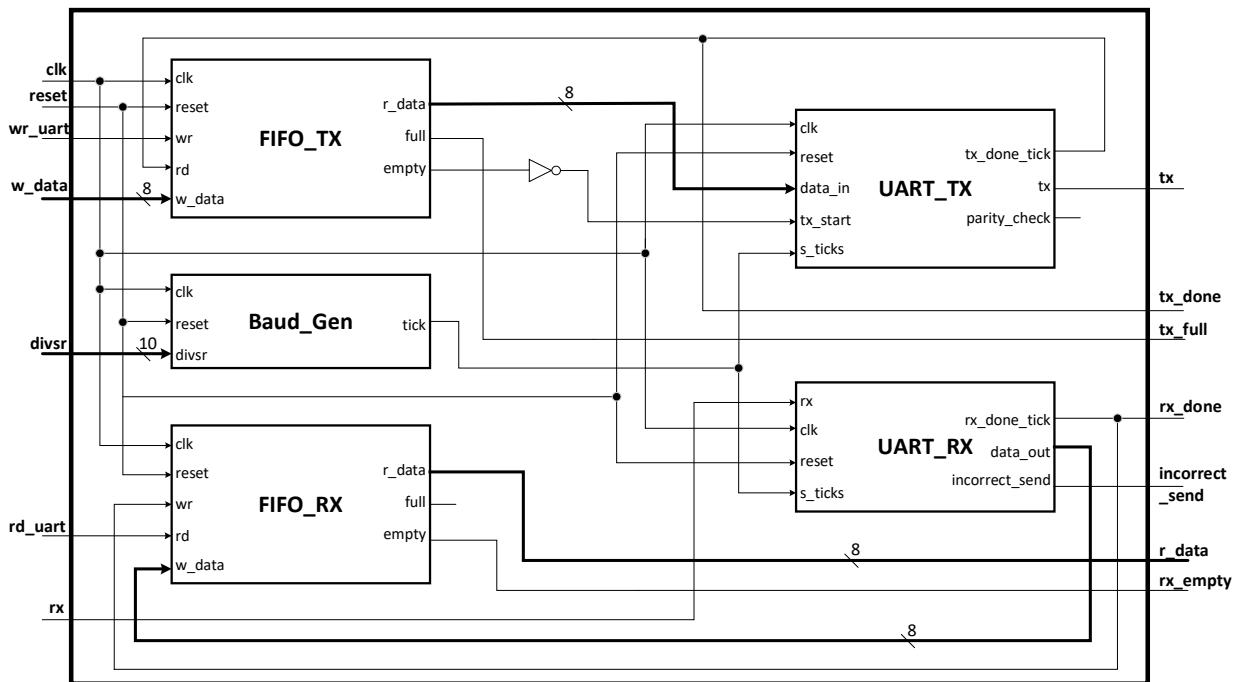
#### 3.1.2. Yêu cầu của thiết kế môi trường UVM

Thiết kế môi trường UVM gồm các yêu cầu:

- Xây dựng môi trường UVM hoàn chỉnh (agent, driver, monitor, sequencer, scoreboard, coverage).
- Sinh dữ liệu kiểm thử ngẫu nhiên có ràng buộc để kiểm tra toàn bộ chức năng UART.
- Ứng dụng functional coverage để theo dõi mức độ bao phủ các chức năng truyền/nhận, trạng thái FIFO và tín hiệu báo lỗi.

### 3.2. THIẾT KẾ IP UART

#### 3.2.1. Tổng quan về thiết kế



Hình 3.1. Sơ đồ khái niệm của thiết kế IP UART

Dưới đây là bảng 3.1 mô tả các chân của sơ đồ khái niệm hình 3.1:

Bảng 3.1: Mô tả các chân của thiết kế IP UART

Chân	Chức năng	Mô tả
clk	Ngõ vào	Xung clock hệ thống cho toàn bộ các khối UART
Reset	Ngõ vào	Tín hiệu reset, đưa hệ thống về trạng thái ban đầu
wr_uart	Ngõ vào	Tín hiệu ghi dữ liệu vào FIFO_TX (yêu cầu truyền)
w_data[7:0]	Ngõ vào	Dữ liệu 8-bit ghi vào FIFO_TX để truyền
divsr[9:0]	Ngõ vào	Hệ số chia clock để tạo nhịp baud/tick cho truyền-nhận
rd_uart	Ngõ vào	Tín hiệu đọc dữ liệu từ FIFO_RX (đọc dữ liệu nhận được)

rx	Ngõ vào	Tín hiệu nhận nối tiếp (Serial input)
tx	Ngõ ra	Tín hiệu truyền nối tiếp (Serial output)
tx_done	Ngõ ra	Báo đã truyền xong 1 frame/1 byte dữ liệu
tx_full	Ngõ ra	Báo FIFO_TX đang đầy (không thể ghi thêm)
r_data[7:0]	Ngõ ra	Dữ liệu 8-bit đọc ra từ FIFO_RX
rx_empty	Ngõ ra	Báo FIFO_RX đang rỗng (không có dữ liệu để đọc)
rx_done	Ngõ ra	Báo đã nhận xong 1 frame/1 byte dữ liệu
incorrect_send	Ngõ ra	Báo lỗi khung/kiểm tra nhận (liên quan dữ liệu nhận không hợp lệ)

**Hình 3.1** mô tả sơ đồ khối RTL của thiết kế UART được sử dụng trong đề tài. Thiết kế UART có nhiệm vụ truyền/nhận dữ liệu nối tiếp bất đồng bộ, sử dụng dữ liệu song song độ rộng 8-bit và được tổ chức thành 5 khối chính: **FIFO\_TX**, **Baud\_Gen**, **UART\_TX**, **UART\_RX** và **FIFO\_RX**. Trong đó, hai khối FIFO đóng vai trò đệm dữ liệu nhằm hạn chế mất dữ liệu khi tốc độ đọc/ghi từ phía điều khiển không kịp tốc độ truyền/nhận trên đường nối tiếp.

- **Khối tạo nhịp Baud\_Gen:**

Khối **Baud\_Gen** nhận *clk*, *reset* và tham số cấu hình *divsr[9:0]* (bus 10-bit) để tạo ra tín hiệu *tick* (hay *s\_ticks*) làm nhịp thời gian cho cả bộ phát và bộ thu. Tham số *divsr* quyết định hệ số chia của clock hệ thống, từ đó xác định tốc độ baud. Tín hiệu *tick* được đưa đồng thời đến **UART\_TX** và **UART\_RX** để đảm bảo quá trình dịch bit và lấy mẫu dữ liệu diễn ra đúng thời điểm.

- **Luồng truyền dữ liệu (Transmit Path):**

Luồng truyền bắt đầu khi người dùng kích hoạt tín hiệu *wr\_uart* để ghi dữ liệu *w\_data[7:0]* (bus 8-bit) vào **FIFO\_TX**. Khối **FIFO\_TX** lưu trữ dữ liệu chờ truyền và cung cấp các tín hiệu trạng thái *full* và *empty*. Từ **FIFO\_TX**, dữ liệu đầu ra *r\_data[7:0]* được đưa vào cổng *data\_in* của khối **UART\_TX**. Đồng thời, tín hiệu *empty* của **FIFO\_TX** được đưa qua một cổng đảo để tạo tín hiệu *tx\_start = ~empty*.

Nghĩa là khi **FIFO\_TX** không rỗng, **UART\_TX** sẽ được cho phép bắt đầu truyền frame. Trong quá trình hoạt động, **UART\_TX** sử dụng *s\_ticks* để tuần tự phát start bit, các data bit, (tùy thiết kế có thể kiểm tra parity qua *parity\_check*) và stop bit ra chân *tx*. Khi truyền xong một frame, **UART\_TX** phát xung *tx\_done\_tick*. Xung này được sử dụng cho hai mục đích:

- Đưa ra ngoài dưới dạng tín hiệu *tx\_done* nhằm báo truyền hoàn tất.
- Hồi tiếp về **FIFO\_TX** để kích hoạt tín hiệu *rd*, giúp **FIFO\_TX** dịch sang byte tiếp theo (nếu còn dữ liệu).

Ngoài ra, tín hiệu *full* của **FIFO\_TX** được đưa ra ngoài dưới tên *tx\_full*, dùng để cảnh báo trạng thái **FIFO\_TX** đầy và không thể ghi thêm dữ liệu mới.

- **Luồng nhận dữ liệu (Receive Path):**

Tín hiệu nối tiếp *rx* được đưa trực tiếp vào khói **UART\_RX**. **UART\_RX** dùng nhịp *s\_ticks* để phát hiện start bit, lấy mẫu các bit dữ liệu và tái tạo dữ liệu song song *data\_out[7:0]*. Khi nhận hoàn tất một frame, **UART\_RX** phát xung *rx\_done\_tick*, xung này đồng thời:

- Được đưa ra ngoài dưới dạng tín hiệu *rx\_done* để báo đã nhận xong.
- Kích hoạt tín hiệu *wr* của **FIFO\_RX** để ghi dữ liệu *data\_out[7:0]* vào **FIFO\_RX**.

**FIFO\_RX** đóng vai trò bộ đệm cho dữ liệu nhận. Khi người dùng kích *rd\_uart*, **FIFO\_RX** xuất dữ liệu ra cổng *r\_data[7:0]*. Tín hiệu *empty* của **FIFO\_RX** được đưa ra ngoài dưới tên *rx\_empty*, báo trạng thái rỗng (không có dữ liệu để đọc). Ngoài ra, **UART\_RX** còn xuất tín hiệu *incorrect\_send* để báo trường hợp nhận không hợp lệ.

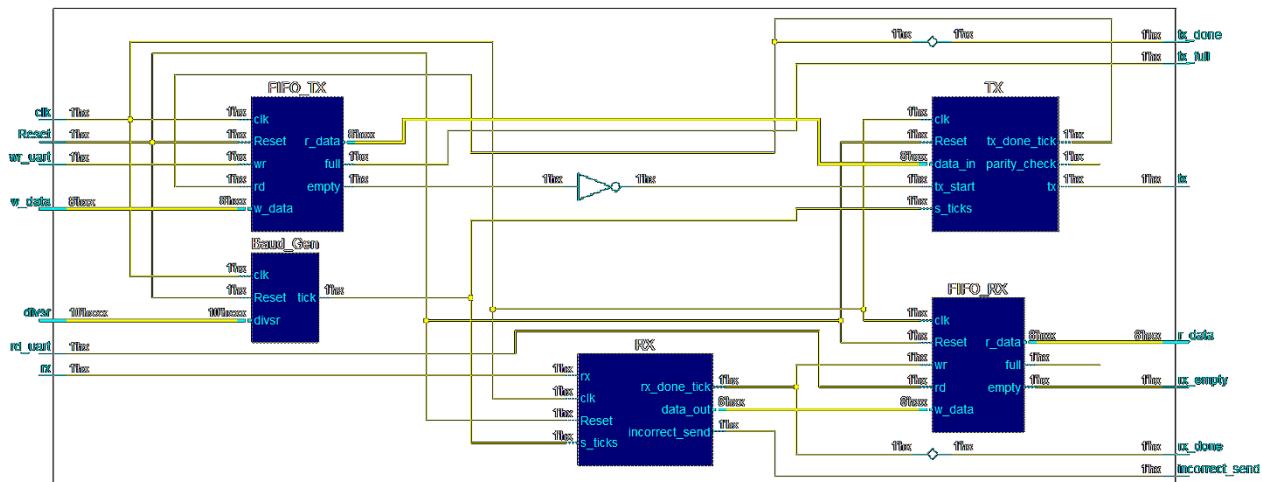
- **Mô tả các đường bus và tín hiệu trạng thái**

- *w\_data[7:0]*, *r\_data[7:0]*, *data\_in[7:0]*, *data\_out[7:0]* là các bus dữ liệu 8-bit, phục vụ truyền/nhận từng byte.
- *divsr[9:0]* là bus cấu hình 10-bit cho bộ tạo baud, dùng để điều chỉnh tốc độ *tick*.
- Các tín hiệu trạng thái quan trọng gồm *tx\_full* (**FIFO\_TX** đầy), *rx\_empty* (**FIFO\_RX** rỗng), *tx\_done/rx\_done* (báo hoàn tất truyền/nhận), và *incorrect\_send* (báo lỗi trong quá trình nhận).

Bảng 3.2: Mô tả chức năng hoạt động của các tín hiệu điều khiển chính

Tín hiệu	Vai trò	Hoạt động
wr_uart	Ghi TX	Khi $wr\_uart=1 \rightarrow$ ghi $w\_data[7:0]$ vào FIFO_TX
tx_start	Kích TX	$tx\_start = \sim empty(FIFO\_TX) \rightarrow$ FIFO còn dữ liệu thì UART_TX bắt đầu truyền
tx_done_tick	Đồng bộ TX	UART_TX phát xung khi truyền xong $\rightarrow$ tạo tx_done và kích rd FIFO_TX
rd_uart	Đọc RX	Khi $rd\_uart=1 \rightarrow$ đọc $r\_data[7:0]$ từ FIFO_RX
rx_done_tick	Đồng bộ RX	UART_RX phát xung khi nhận xong $\rightarrow$ tạo rx_done và kích wr FIFO_RX
divsr[9:0]	Cấu hình baud	Đặt hệ số chia cho Baud_Gen $\rightarrow$ tạo tick/s_ticks cho TX/RX

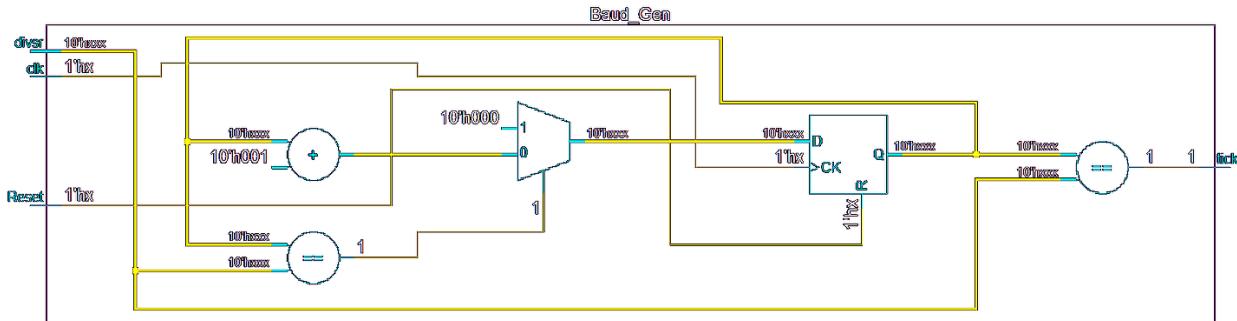
### 3.2.2. Sơ đồ RTL của thiết kế IP UART



Hình 3.2. Sơ đồ RTL của thiết kế UART trên QuestaSim

Hình 3.2 thể hiện sơ đồ RTL của thiết kế IP UART được tạo ra sau khi mô tả bằng SystemVerilog và quan sát trên QuestaSim. Thiết kế bao gồm 5 khối chính: **FIFO\_TX**, **Baud\_Gen**, **UART\_TX**, **UART\_RX** và **FIFO\_RX**. Tương tự như sơ đồ khối được vẽ ở hình 3.1 trước đó.

### 3.2.3. Khối Baud\_Rate\_Gen

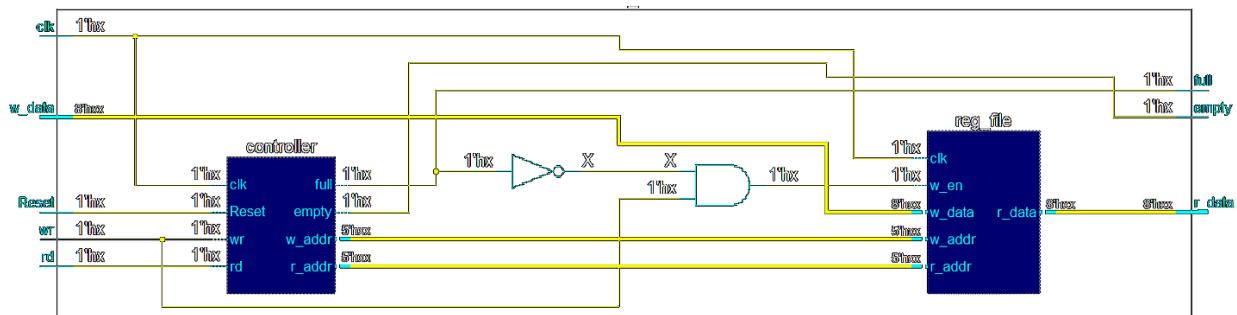


Hình 3.3. Sơ đồ RTL của khói Baud\_Gen

**Hình 3.3** mô tả cấu tạo và nguyên lý hoạt động của khói **Baud\_Gen** (Baud rate generator) trong thiết kế UART. Khối này có các ngõ vào gồm *clk*, *Reset* và *divsr[9:0]*, trong đó *divsr[9:0]* là giá trị cấu hình hệ số chia để tạo tốc độ baud mong muốn. Ngõ ra của khói là tín hiệu *tick* (hay *s\_ticks*) đóng vai trò nhịp thời gian dùng chung cho **UART\_TX** và **UART\_RX** nhằm đồng bộ quá trình truyền và lấy mẫu dữ liệu.

Về cơ chế hoạt động, **Baud\_Gen** sử dụng một bộ đếm nội *count\_logic* tăng dần theo từng cạnh lên của *clk*. Ở mỗi chu kỳ, bộ đếm sẽ được so sánh với giá trị *divsr[9:0]*. Khi *count\_logic* == *divsr*, khói sẽ phát một xung *tick* = 1 trong đúng một chu kỳ clock, đồng thời đưa bộ đếm về 0 để bắt đầu chu kỳ chia mới. Nếu *count\_logic* chưa đạt đến *divsr*, bộ đếm tiếp tục tăng *count\_logic* = *count\_logic* + 1 và *tick* giữ ở mức 0. Khi *Reset* được kích hoạt, toàn bộ trạng thái của bộ đếm được đưa về 0 và *tick* cũng được đặt về 0, đảm bảo hệ thống khởi tạo lại đúng trạng thái ban đầu.

### 3.2.4. Khối FIFO (bao gồm cả FIFO\_RX và TX)



Hình 3.4. Sơ đồ RTL rút gọn của khói FIFO

**Hình 3.4** mô tả sơ đồ RTL rút gọn của khối **FIFO** trong thiết kế UART (áp dụng tương tự cho **FIFO\_TX** và **FIFO\_RX**). Khối FIFO có nhiệm vụ **đệm dữ liệu 8-bit** nhằm tách biệt tốc độ ghi/đọc của hệ thống điều khiển với tốc độ truyền/nhận của UART, qua đó hạn chế mất dữ liệu khi quá trình xử lý của CPU/host không theo kịp tốc độ truyền thông nối tiếp.

Về cấu trúc, FIFO được xây dựng từ hai phần chính:

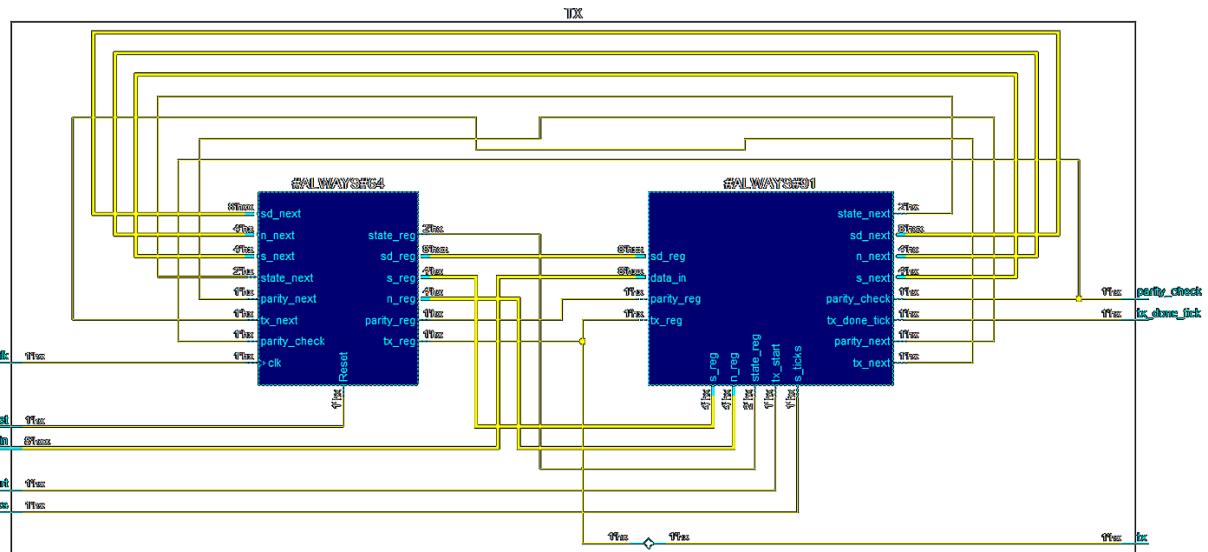
- **Khối controller (FIFO\_Contr):** nhận các tín hiệu *clk*, *Reset*, *wr*, *rd* và tạo ra các tín hiệu điều khiển/trạng thái gồm *full*, *empty*. Đồng thời, controller sinh ra địa chỉ ghi/đọc *w\_addr* và *r\_addr* để điều khiển truy cập bộ nhớ, và phát tín hiệu cho phép ghi *w\_en* khi điều kiện ghi hợp lệ.
- **Khối reg\_file (Register file / memory):** đóng vai trò vùng lưu trữ dữ liệu. Khi *w\_en=1*, dữ liệu *w\_data[7:0]* được ghi vào ô nhớ tại địa chỉ *w\_addr*. Dữ liệu đọc ra được truy xuất theo địa chỉ *r\_addr* và xuất ra cổng *r\_data[7:0]*.

#### Nguyên lý hoạt động của FIFO như sau:

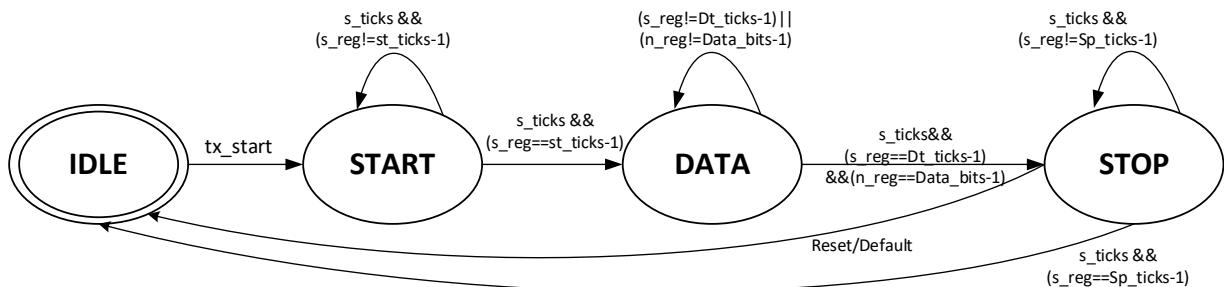
Khi *wr=1* và FIFO chưa đầy (*full=0*), controller cho phép ghi (*w\_en=1*), dữ liệu *w\_data* được lưu vào *reg\_file* tại địa chỉ *w\_addr*, đồng thời con trỏ ghi được cập nhật sang vị trí kế tiếp. Ngược lại, khi *rd=1* và FIFO không rỗng (*empty=0*), controller cập nhật con trỏ đọc để xuất byte tiếp theo ra *r\_data*. Hai tín hiệu *full* và *empty* được cập nhật dựa trên quan hệ giữa con trỏ ghi và con trỏ đọc, nhờ đó hệ thống có thể biết khi nào FIFO không thể ghi thêm hoặc không còn dữ liệu để đọc.

Trong thiết kế UART, **FIFO\_TX** dùng để lưu dữ liệu chờ truyền từ *w\_data[7:0]* sang **UART\_TX**, còn **FIFO\_RX** dùng để lưu dữ liệu nhận từ **UART\_RX** trước khi đưa ra cổng *r\_data[7:0]*. Việc sử dụng FIFO ở cả hai nhánh TX/RX giúp thiết kế hoạt động ổn định, tăng khả năng chịu tải và hỗ trợ truyền/nhận liên tục theo đúng thứ tự “vào trước – ra trước”.

### 3.2.5. Khởi UART\_TX (UART Transmitter)



Hình 3.5. Sơ đồ RTL rút gọn của khối UART TX



Hình 3.6. Sơ đồ FSM trong thiết kế UART TX

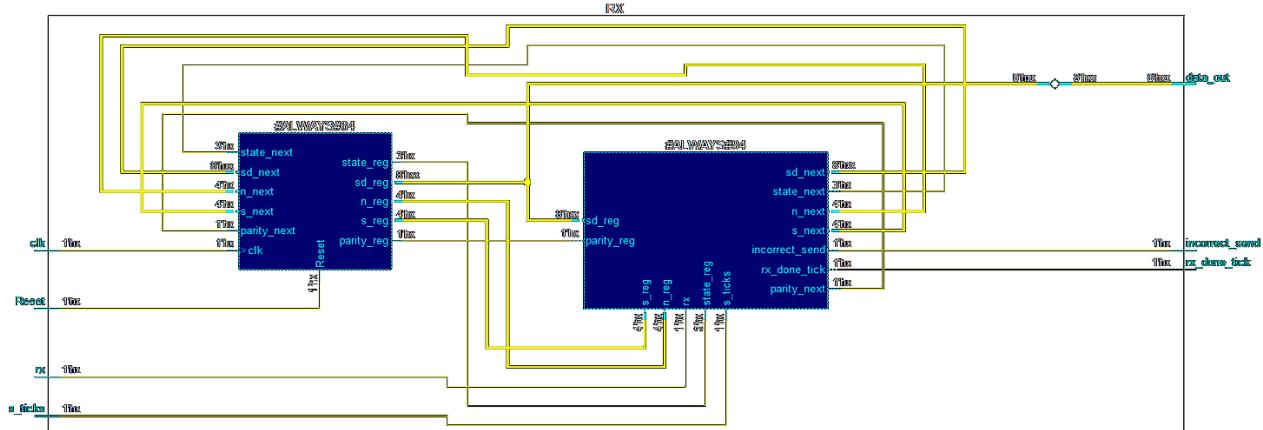
**Hình 3.5** mô tả sơ đồ RTL rút gọn của khối **UART\_TX**, đây là khối đảm nhiệm chức năng **truyền dữ liệu nối tiếp** ra chân *tx* theo chuẩn UART. Khối **UART\_TX** nhận các tín hiệu vào gồm *clk*, *Reset*, dữ liệu song song *data\_in[7:0]*, tín hiệu khởi phát *tx\_start* và xung thời gian *s\_ticks* từ khối **Baud\_Gen**. Các tín hiệu ngõ ra chính bao gồm *tx* (đường truyền nối tiếp), *tx\_done\_tick* ( báo truyền xong 1 frame) và *parity\_check* (phục vụ kiểm tra/giám sát tính chẵn lẻ theo thiết kế).

Về nguyên lý hoạt động, **UART\_TX** được tổ chức theo mô hình FSM (Finite State Machine) như thể hiện ở **Hình 3.6**, gồm 4 trạng thái chính:

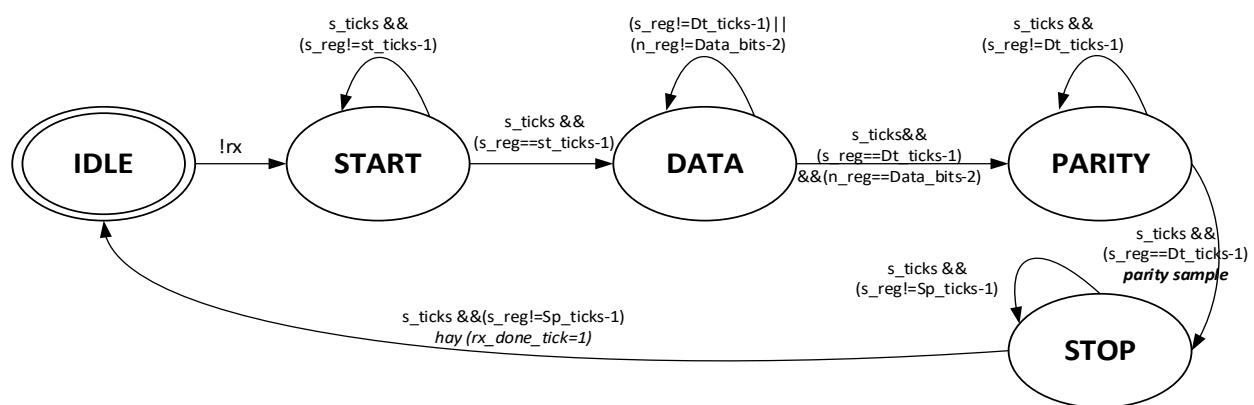
- **IDLE**: Trạng thái chờ. Ở trạng thái này *tx* được giữ ở mức “1”. Khi *tx\_start* được kích hoạt, FSM chuyển sang trạng thái START để bắt đầu truyền.

- START:** Phát start bit (thường là mức “0”) trong một khoảng thời gian xác định theo nhịp  $s\_ticks$ . Trong quá trình ở START, bộ đếm  $s\_reg$  được sử dụng để đếm số tick, khi đạt ngưỡng (ở đây  $s\_reg == St\_ticks-1$ ) thì chuyển sang trạng thái DATA.
- DATA:** Truyền lần lượt các bit dữ liệu của  $data\_in[7:0]$ . Khối sử dụng bộ đếm tick  $s\_reg$  để giữ mỗi bit dữ liệu đúng thời lượng baud, đồng thời dùng bộ đếm bit  $n\_reg$  để xác định đã truyền đến bit thứ bao nhiêu. Khi chưa truyền đủ toàn bộ số bit dữ liệu, FSM tiếp tục lặp trong trạng thái DATA; khi điều kiện kết thúc dữ liệu thỏa (ở đây  $s\_reg == Dt\_ticks-1$  và  $n\_reg == Data\_bits-1$ ) thì chuyển sang STOP.
- STOP:** Phát stop bit (thường là mức “1”) trong thời lượng quy định. Khi kết thúc stop bit (ở đây  $s\_reg == Sp\_ticks-1$ ), **UART\_TX** phát xung  $tx\_done\_tick$  để báo hoàn tất truyền 1 frame, sau đó quay về trạng thái IDLE để chờ lần truyền tiếp theo.

### 3.2.6. Khối UART\_RX (UART Receiver)



Hình 3.7. Sơ đồ RTL rút gọn của khối UART\_RX



Hình 3.8. Sơ đồ FSM trong thiết kế UART\_RX

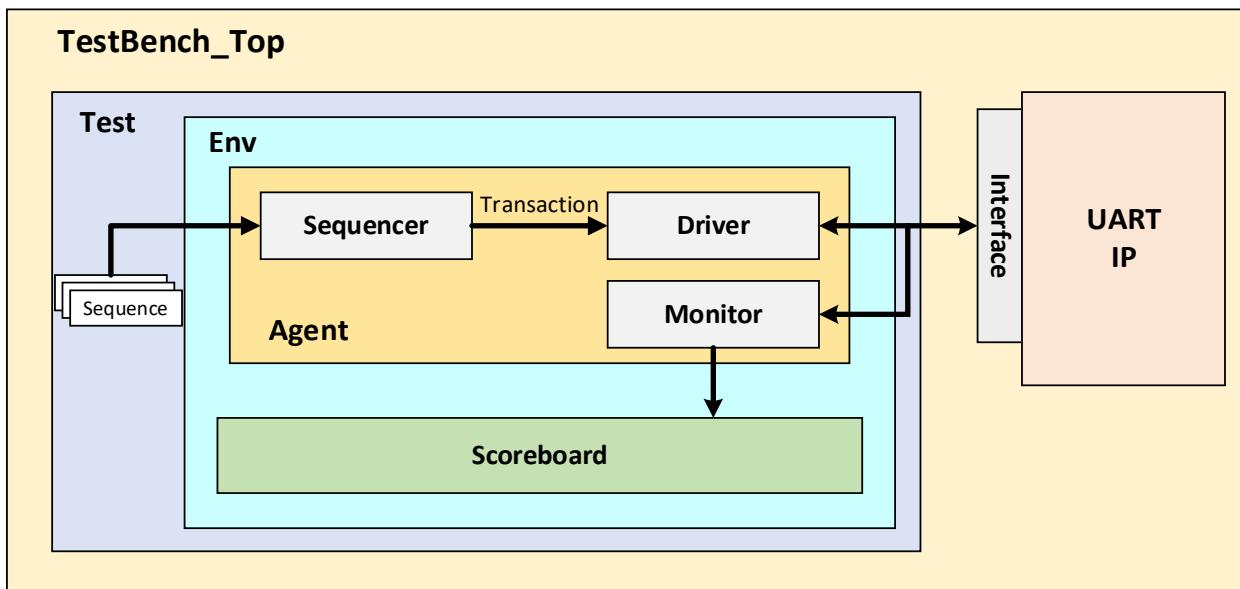
**Hình 3.7** mô tả sơ đồ RTL rút gọn của khối **UART\_RX**, đây là khối đảm nhiệm chức năng thu dữ liệu nối tiếp từ chân *rx* và chuyển đổi sang dữ liệu song song *data\_out[7:0]*. **UART\_RX** nhận các tín hiệu vào gồm *clk*, *Reset*, tín hiệu nối tiếp *rx* và xung thời gian *s\_ticks* từ khối **Baud\_Gen** để đồng bộ quá trình lấy mẫu. Các tín hiệu ngõ ra chính bao gồm *data\_out[7:0]* (dữ liệu nhận được), *rx\_done\_tick* (báo nhận xong 1 frame) và *incorrect\_send* (báo lỗi/không hợp lệ trong quá trình nhận theo logic thiết kế).

Nguyên lý hoạt động của **UART\_RX** được điều khiển bởi FSM như thể hiện ở **Hình 3.8**, gồm 5 trạng thái chính:

- **IDLE**: Trạng thái chờ. Ở trạng thái này **UART\_RX** giám sát đường *rx*. Khi phát hiện *rx* chuyển xuống mức “0” (điều kiện *!rx*) thì nhận biết có start bit, FSM chuyển sang trạng thái **START** để bắt đầu quá trình thu.
- **START**: Ở trạng thái này **UART\_RX** thực hiện căn chỉnh thời điểm lấy mẫu bằng cách đếm *s\_ticks* thông qua thanh ghi *s\_reg*. Khi đạt đến ngưỡng xác định (ở đây *s\_reg == St\_ticks-1*), hệ thống chuyển sang trạng thái **DATA** để bắt đầu lấy mẫu các bit dữ liệu.
- **DATA**: Trạng thái thu dữ liệu. **UART\_RX** lấy mẫu từng bit dữ liệu theo nhịp *s\_ticks*, mỗi bit được giữ trong thời lượng *Dt\_ticks*. Bộ đếm *s\_reg* dùng để đếm tick trong một bit, còn *n\_reg* dùng để đếm số bit dữ liệu đã thu. Khi chưa đủ số bit dữ liệu, FSM tiếp tục lặp trong **DATA**. Khi đạt điều kiện kết thúc phần dữ liệu (ở đây *s\_reg == Dt\_ticks-1* và *n\_reg == Data\_bits-2*), FSM chuyển trạng thái **PARITY**.
- **PARITY**: Trạng thái kiểm tra/parity. **UART\_RX** lấy mẫu bit *parity* tại thời điểm thích hợp theo *s\_ticks* (parity sample), đồng thời cập nhật các thanh ghi liên quan và có thể tạo điều kiện xác định lỗi. Khi hoàn tất thời gian *parity* (ở đây *s\_reg == Dt\_ticks-1*), FSM chuyển sang trạng thái **STOP**.
- **STOP**: Ở trạng thái này **UART\_RX** lấy mẫu stop bit (mức “1”) trong khoảng *Sp\_ticks*. Khi kết thúc stop bit (ở đây *s\_ticks && s\_reg == Sp\_ticks-1*), **UART\_RX** phát xung *rx\_done\_tick* để báo nhận xong một frame, đồng thời đưa FSM quay lại **IDLE** để sẵn sàng nhận byte tiếp theo. Nếu trong quá trình kiểm tra phát hiện dữ liệu không hợp lệ, tín hiệu *incorrect\_send* sẽ được kích hoạt nhằm cảnh báo lỗi.

Tương tự khôi phát, khi *Reset* được kích hoạt, FSM **UART\_RX** quay về **IDLE** và các thanh ghi đếm (*s\_reg*, *n\_reg*...), cũng như các tín hiệu điều khiển sẽ được đưa về giá trị mặc định. Nhờ việc điều khiển bằng FSM và lấy mẫu theo *s\_ticks*, khối **UART\_RX** đảm bảo việc thu dữ liệu nối tiếp và chuyển đổi sang dữ liệu song song diễn ra ổn định, đúng định dạng UART theo cấu hình thiết kế.

### 3.3. THIẾT KẾ MÔI TRƯỜNG UVM



Hình 3.9. Sơ đồ khái niệm khung mô hình kiểm tra UVM

**Hình 3.9** thể hiện sơ đồ khái niệm khung mô hình kiểm tra UVM dùng để kiểm tra thiết kế **UART IP**. Mô hình được tổ chức theo cấu trúc phân cấp chuẩn của UVM gồm **TestBench\_Top**, **Test**, **Env**, **Agent** (bao gồm Sequencer – Driver – Monitor), **Scoreboard** và **Interface** kết nối trực tiếp với DUT (UART IP). Các giao dịch kiểm thử (transaction) được tạo bởi **Sequence**, đi qua **Sequencer** đến **Driver** để kích thích DUT, đồng thời **Monitor** thu thập phản hồi từ DUT và gửi dữ liệu về **Scoreboard** để đối chiếu/đánh giá.

**Nguyên lý hoạt động:** hệ thống UVM được khởi tạo và hoạt động theo ba pha:

- **Pha khởi tạo:** Mô hình UVM được xây dựng theo cấu trúc phân cấp, khởi tạo các thành phần trong **Env/Agent/Scoreboard**. Đồng thời, virtual interface được cấu hình để liên kết mô hình UVM với **UART interface** của DUT, đảm bảo Driver/Monitor có thể truy cập các tín hiệu phần cứng.

- **Pha thực thi:**

- **Sequence** tạo ra các transaction (ví dụ: ghi dữ liệu truyền *wr\_uart + w\_data*, đọc dữ liệu nhận *rd\_uart*, cấu hình *divsr*, tạo các tình huống RX/TX...).
- **Sequencer** điều phối và cấp transaction cho **Driver**.
- **Driver** chuyển transaction thành tín hiệu mức chân (pin-level) thông qua interface và kích thích **UART IP** theo đúng thời điểm clock/reset.
- **Monitor** quan sát các tín hiệu trên interface (TX/RX, FIFO status, *tx\_done/rx\_done*, *r\_data...*) và đóng gói lại thành transaction rồi phát về **Scoreboard** qua công phân tích (analysis/TLM).

Tại **Scoreboard**, dữ liệu thực tế do Monitor thu được sẽ được so sánh với **giá trị mong đợi** (expected). Với UART, so khớp thường tập trung vào: byte truyền ra đường *tx* có đúng dữ liệu đã push vào FIFO\_TX không; byte nhận được ở *r\_data* có khớp với chuỗi kích thích ở đường *rx* không; các cờ trạng thái *tx\_full*, *rx\_empty*, *tx\_done*, *rx\_done*, *incorrect\_send* có bật đúng tình huống hay không. Nếu kết quả trùng khớp thì testcase đạt, ngược lại Scoreboard sẽ báo lỗi.

- **Pha kết thúc:** Sau khi hoàn thành mô phỏng, hệ thống tổng hợp kết quả kiểm thử, thống kê lỗi/cảnh báo và xuất log/biểu đồ dạng sóng (waveform) phục vụ phân tích và gỡ lỗi. Khi tất cả testcase đạt yêu cầu, quá trình mô phỏng kết thúc.

#	Name	Type	Size	Value
#	uvm_test_top	test	-	@472
#	UART_env	env	-	@483
#	UART_agent	agent	-	@490
#	UART_driver	driver	-	@645
#	rsp_port	uvm_analysis_port	-	@660
#	seq_item_port	uvm_seq_item_pull_port	-	@652
#	UART_monitor	monitor	-	@521
#	monitor_port	uvm_analysis_port	-	@528
#	UART_sequencer	sequencer	-	@536
#	rsp_export	uvm_analysis_export	-	@543
#	seq_item_export	uvm_seq_item_pull_imp	-	@637
#	arbitration_queue	array	0	-
#	lock_queue	array	0	-
#	num_last_reqs	integral	32	'd1
#	num_last_rsps	integral	32	'd1
#	UART_coverage_collector	coverage_collector	-	@504
#	analysis_imp	uvm_analysis_imp	-	@511
#	coverage_collector_in_imp	uvm_analysis_imp	-	@675
#	UART_scoreboard	scoreboard	-	@497
#	scoreboard_imp	uvm_analysis_imp	-	@683

Hình 3.10. UVM topology của môi trường kiểm thử UART

**Hình 3.10** thể hiện **cấu trúc phân cấp (UVM topology)** của môi trường kiểm thử UART được in ra trực tiếp từ quá trình mô phỏng bằng lệnh `uvm_top.print_topology()`. Thông qua topology này có thể xác nhận rằng các thành phần chính của testbench đã được khởi tạo đúng theo thiết kế, bao gồm `uvm_test_top` chứa `UART_env`, bên trong có `UART_agent` với đầy đủ ba thành phần **UART\_driver**, **UART\_monitor** và **UART\_sequencer**. Cùng với **UART\_scoreboard** để kiểm tra đúng/sai của dữ liệu và **UART\_coverage\_collector** để thu thập coverage trong quá trình mô phỏng.

Bên cạnh việc mô tả tên và loại component, topology còn cho thấy các kênh kết nối TLM/analysis giữa các thành phần, ví dụ monitor phát transaction qua `monitor_port` và được nhận bởi các khối phân tích như scoreboard/coverage thông qua `analysis_export` hoặc các analysis implementation port tương ứng. Nhờ đó, dữ liệu quan sát từ DUT có thể được truyền về scoreboard để so sánh với kết quả mong đợi, đồng thời được ghi nhận cho mục tiêu coverage.

Tóm lại, việc in topology không chỉ giúp **kiểm tra nhanh môi trường UVM đã build đúng cấu trúc phân cấp**, mà còn là cơ sở chứng minh mô hình kiểm thử UART được triển khai đầy đủ các khía cạnh thiết, hỗ trợ tốt cho việc debug và đánh giá kết quả mô phỏng.

### 3.4. XÂY DỰNG KẾ HOẠCH KIỂM TRA

Mục tiêu của kế hoạch kiểm tra là tạo kích thích và quan sát để xác minh đầy đủ các chức năng chính của UART IP, bao gồm: **Reset**, **đường truyền TX**, **đường nhận RX**, **FIFO TX/RX**, **các cờ trạng thái** (`tx_full`, `rx_empty`, `tx_done`, `rx_done`) và cờ lỗi `incorrect_send`. Các testcase được triển khai dưới dạng sequence và được đánh giá thông qua **Scoreboard** (so khớp dữ liệu mong đợi – thực tế) đồng thời thu thập **functional coverage** trong **coverage\_collector**.

#### 3.4.1. Kiểm tra Reset hệ thống

Thực hiện testcase reset bằng `reset_sequence`. Driver sẽ áp reset trong một số chu kỳ clock (theo `apply_reset()`), đưa DUT về trạng thái ban đầu. Monitor đồng thời phát transaction `TR_RESET` để ghi nhận sự kiện reset vào coverage (coverpoint `KIND, RESET`) và giúp scoreboard xóa hàng đợi dữ liệu mong đợi để tránh sai lệch sau reset.

### 3.4.2. Kiểm tra chức năng truyền – nhận cơ bản (Write → Read)

Sử dụng các sequence dạng **write + read** (lớp *single\_wr\_rd\_sequence* và các lớp kế thừa) để kiểm tra luồng cơ bản:

- *all\_zero\_sequence* (8'h00), *all\_one\_sequence* (8'hFF), *pat\_aa\_sequence* (8'hAA), *pat\_55\_sequence* (8'h55).
- Driver thực hiện *wr\_uart* để đẩy *w\_data* vào FIFO\_TX, sau đó chờ RX có dữ liệu (*rx\_empty==0*) và kích *rd\_uart* để đọc ra *r\_data*.
- Scoreboard sẽ **push** dữ liệu kỳ vọng khi thấy *TR\_WRITE* và **pop+compare** khi thấy *TR\_READ*. Nếu *r\_data* khớp *w\_data* thì testcase đạt, ngược lại báo lỗi.
- Các testcase này đồng thời nhắm phủ các bin quan trọng của coverage: *W\_DATA* và *R\_DATA* (00/FF/AA/55).

### 3.4.3. Kiểm tra trạng thái FIFO\_TX đầy (tx\_full) và hành vi ghi/đọc theo burst

Sử dụng *fill\_tx\_full\_sequence* để tạo tình huống fill FIFO\_TX đến ngưỡng đầy:

- Giai đoạn 1: burst **write-only** liên tục *depth* lần để ép trạng thái *tx\_full* chuyển sang 1.
- Giai đoạn 2: burst **read-only** để “drain” dữ liệu (đảm bảo luồng expected trong scoreboard vẫn đồng bộ).
- Testcase này được chạy với nhiều mẫu dữ liệu (00/FF/AA/55) nhằm hit coverage:
  - *TX\_FULL* (not\_full/full)
  - *WDATA\_STATUS* và đặc biệt là cross *CROSS\_WDATA\_FULL* .

### 3.4.4. Kiểm tra đường nhận RX và cơ chế báo lỗi (incorrect\_send)

Sử dụng *rx\_bad\_parity\_sequence* để **inject frame RX** từ testbench qua chân *rx*:

- Sequence có bước cấu hình divider tùy chọn để đảm bảo timing lấy mẫu đúng.
- Sau đó driver phát 1 hoặc nhiều frame lên *rx*, có chủ đích tạo **parity sai** (*bad\_parity=1*) nhằm kích hoạt cờ *incorrect\_send*.
- Monitor sẽ phát transaction *TR\_STATUS* khi có thay đổi các cờ trạng thái, qua đó coverage sẽ ghi nhận các bin:

- *INCORRECT\_SEND* (ok/err)
- *RX\_DONE, TX\_DONE* (low/high nếu quan sát được)
- *DIVSR* (d54/small/mid/big tùy cấu hình)

### 3.4.5. Kiểm tra ngẫu nhiên và stress (random/system-like)

Sử dụng *random\_mix\_sequence* để mô phỏng tình huống “gần thực tế”:

- Trộn hai kiểu hoạt động: (1) giao dịch đơn write+read với *read\_delay\_cycles* ngẫu nhiên, (2) burst write-only rồi burst read-only (độ dài burst giới hạn *max\_burst* để an toàn với FIFO depth).
- Dữ liệu *w\_data* được tạo ngẫu nhiên bằng *\$urandom\_range(0,255)* nhằm phủ *default\_bin\_others* của *W\_DATA/R\_DATA*, đồng thời tạo thêm nhiều chuyển trạng thái cờ *rx\_empty*, *tx\_full*, *tx\_done*, *rx\_done* để tăng coverage tổng thể.

Các tham số có thể điều chỉnh bằng plusargs như *RAND\_TXN*, *RAND\_MAX\_BURST*, *RAND\_MAX\_RD\_DLY*.

## CHƯƠNG 4: KẾT QUẢ VÀ ĐÁNH GIÁ

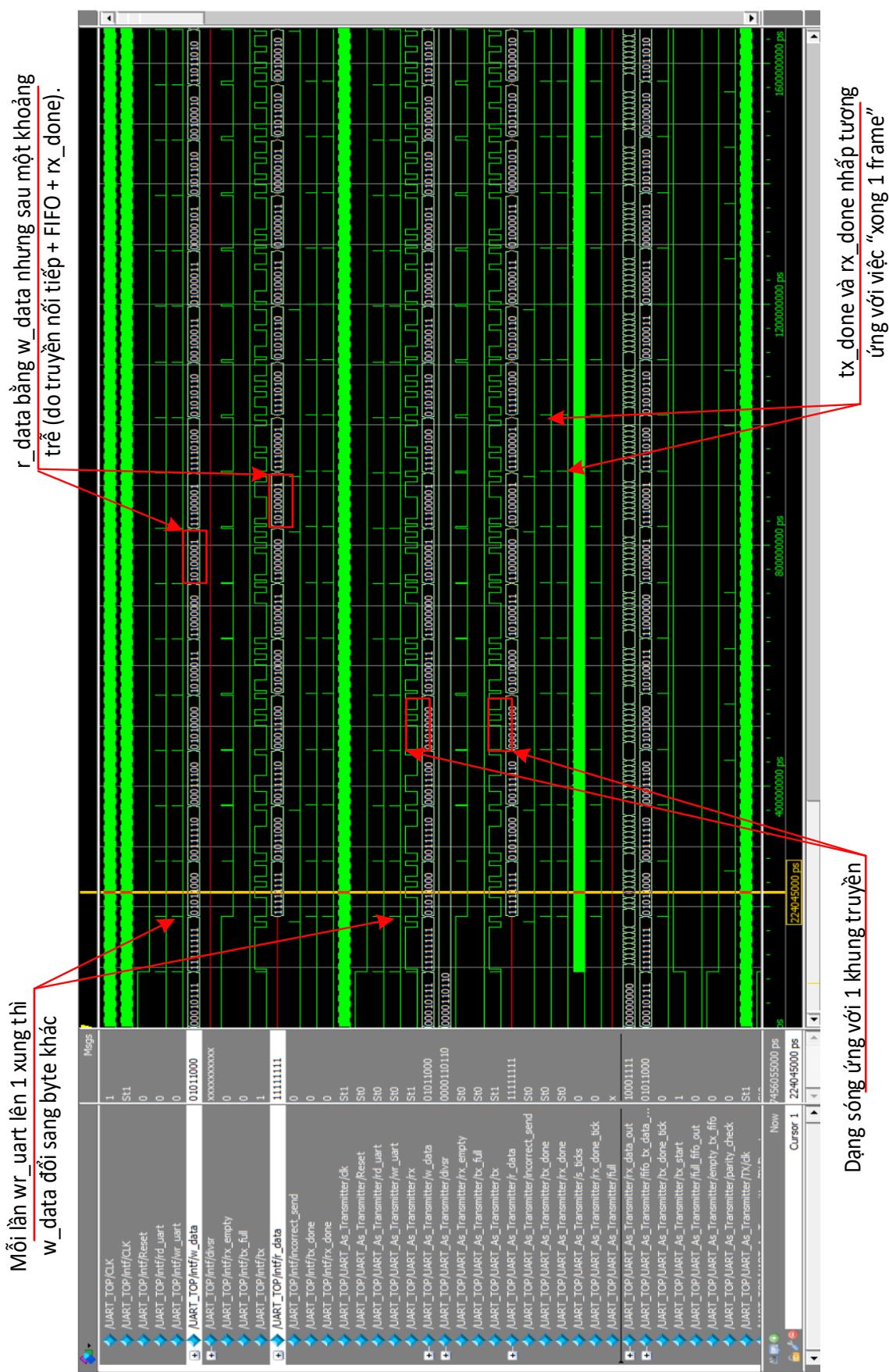
*Trong đồ án này, em chỉ thực hiện mô hình LOOPBACK cho môi trường kiểm tra chức năng UART, nên các waveform kết quả sẽ truyền bits qua chân tx và nhận tại chân rx của chính nó.*

### 4.1. KẾT QUẢ MÔ PHỎNG CHỨC NĂNG TRUYỀN – NHẬN UART (TX/RX)

Quan sát waveform **Hình 4.1** cho thấy môi trường UVM đã tạo stimulus ghi dữ liệu vào TX FIFO bằng *wr\_uart/w\_data*, sau đó dữ liệu được UART **serialize** ra đường *tx*, đồng thời phía nhận **deserialize** và đưa byte nhận được ra *r\_data*. Việc đọc dữ liệu từ **RX FIFO** được thực hiện qua *rd\_uart*. Trong quá trình mô phỏng, các cờ trạng thái *tx\_full*, *rx\_empty*, cùng các xung *tx\_done*, *rx\_done* thể hiện đúng cơ chế bắt tay (handshake) theo luồng TX→RX.

#### Diễn tiến tín hiệu chính theo các mốc thời gian trên waveform

- Giai đoạn khởi tạo (Reset)
  - Khi Reset = 1, DUT ở trạng thái khởi tạo: các thao tác *wr\_uart*, *rd\_uart* không được kích hoạt; đường truyền *tx* ở trạng thái idle = 1 (UART line idle high).
  - Sau khi nhả reset (*Reset = 0*), DUT bắt đầu nhận lệnh ghi/đọc từ driver.
- Giai đoạn ghi dữ liệu vào TX (push TX FIFO) và truyền ra *tx*
  - Driver phát các xung *wr\_uart* để ghi byte vào TX FIFO; bus *w\_data* thay đổi theo từng giao dịch. Trên waveform có thể thấy chuỗi byte ví dụ:
    - *w\_data*: 00010111 (0x17) → 11111111 (0xFF) → 01011000 (0x58)  
→ 00111110 (0x3E) → 00011100 (0x1C) → ...
  - Trong suốt quá trình này, *tx\_full* = 0 (TX FIFO chưa đầy) nên các lần ghi đều được chấp nhận.
- Mốc quan sát rõ tại con trỏ (Cursor ≈ 224.045 ns)
  - Tại thời điểm *t* ≈ 224.045 ns, các tín hiệu thể hiện:
    - Reset = 0, DUT đang hoạt động bình thường.
    - *w\_data* = 01011000 (0x58) (byte chuẩn bị/đang được kích để truyền).
    - *r\_data* = 11111111 (0xFF) (byte phía RX đã nhận được trước đó).



Hình 4.1 – Kết quả mô phỏng chức năng truyền-nhận UART.

- rx\_empty = 0 (RX FIFO đang có dữ liệu chờ đọc)
- tx\_full = 0 (TX FIFO còn chỗ trống)
- incorrect\_send = 0 (không phát hiện lỗi khung/parity trong đoạn này)
  - Việc tại cùng một thời điểm  $w\_data = 0x58$  nhưng  $r\_data = 0xFF$  là bình thường, do luồng UART có độ trễ: **byte được ghi vào TX trước**, sau đó mới truyền/nhận xong để xuất hiện ở  $r\_data$ .
- Giai đoạn nhận và đọc dữ liệu (RX FIFO pop)
  - Sau mỗi khung UART, các xung  $tx\_done/rx\_done$  xuất hiện để báo hoàn tất truyền/nhận.
  - Khi  $rx\_empty$  chuyển 0, driver sẽ kích  $rd\_uart$  để đọc dữ liệu; trên waveform,  $r\_data$  lần lượt hiển thị các giá trị khớp theo thứ tự đã ghi trước đó (ví dụ thấy  $11111111 \rightarrow 01011000 \rightarrow 00111110 \rightarrow \dots$ ), thể hiện **đúng nguyên lý loopback/dòng bộ TX–RX**.

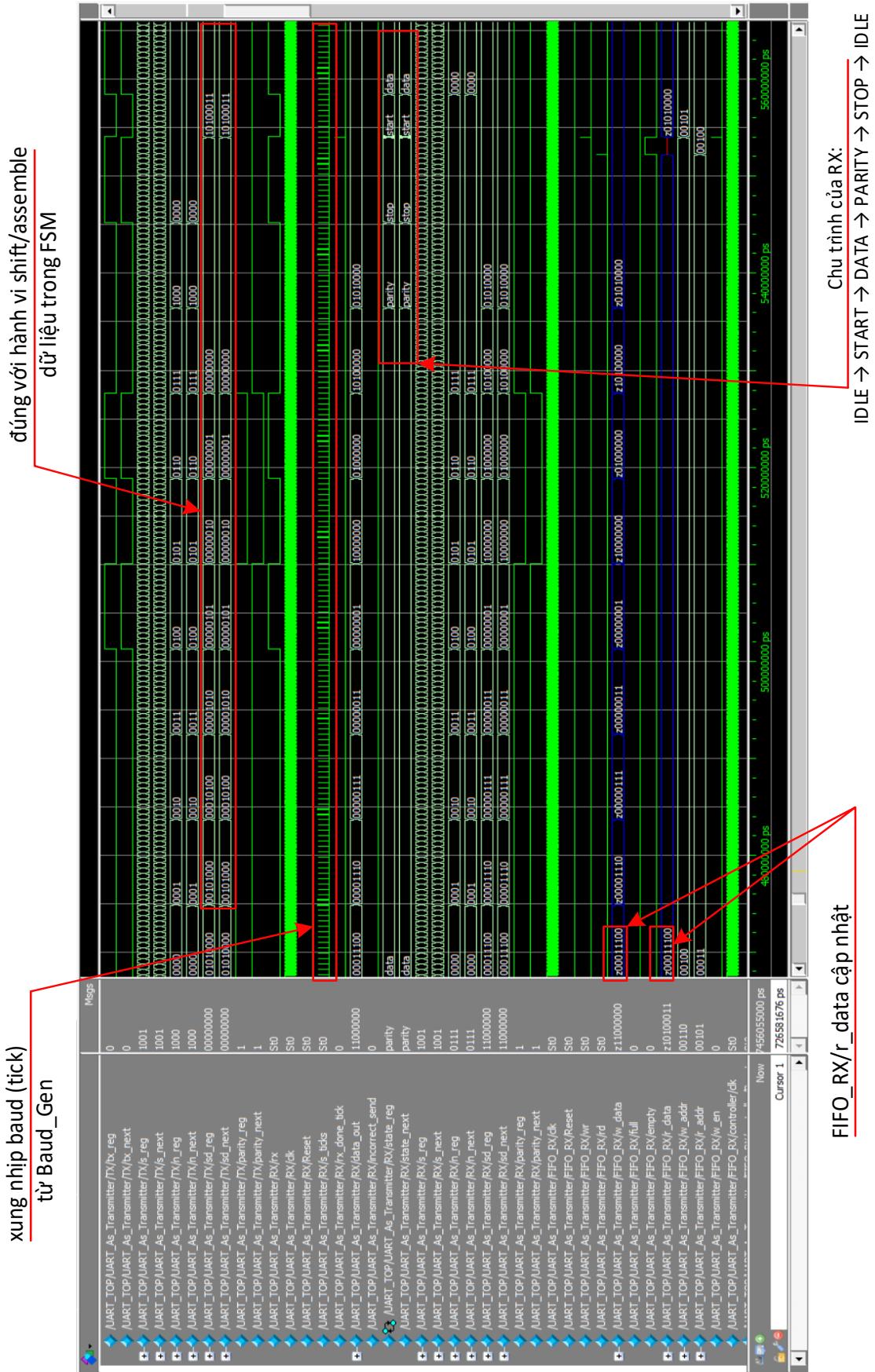
**Kết luận:** Dựa trên Hình 4.1, DUT UART hoạt động đúng luồng Write → Transmit → Receive → Read:  $tx\_full/rx\_empty$  phản ánh đúng trạng thái FIFO,  $tx$  thể hiện truyền nối tiếp,  $r\_data$  xuất hiện theo thứ tự dữ liệu đã phát, và  $incorrect\_send$  giữ 0 trong đoạn mô phỏng này ⇒ chức năng truyền – nhận UART đạt yêu cầu theo kịch bản kiểm tra cơ bản của UVM.

(Một số kịch bản chi tiết được trình bày ở mục 3.4 sẽ không được đánh giá bằng waveform vì phải đánh giá qua timing rất dài nên sẽ chỉ được tóm tắt ngắn qua LOG và coverage report phía sau)

## 4.2. MÔ PHỎNG FSM TX/RX VÀ HÀNH VI FIFO (TX/RX & FIFO Behavior)

Quan sát waveform **Hình 4.2** cho thấy phía RX hoạt động theo FSM với chuỗi trạng thái hiển thị trực tiếp trên RX/state\_reg và RX/state\_next (data → parity → stop → start → data).

Song song đó, dữ liệu sau khi RX thu xong sẽ được đưa vào FIFO\_RX (thể hiện qua  $w\_addr$ ,  $empty$ , và bus  $w\_data$ ), và khi có đọc (FIFO\_RX/rd) thì dữ liệu được xuất ra FIFO\_RX/ $r\_data$  và  $r\_addr$  tăng.



Hình 4.2 – Kết quả mô phỏng FSM của TX/RX và cơ chế đầy/đọc dữ liệu qua FIFO\_RX.

### Diễn tiến hoạt động RX FSM theo waveform

- Giai đoạn RX đang thu DATA (state = data)
  - Trên đường RX/state\_reg và RX/state\_next xuất hiện chữ data kéo dài trong một khoảng.
  - Trong giai đoạn này, bộ đếm bit RX/n\_reg chạy tuần tự 0000 → 0001 → ... → 0111 (đếm đủ 8 bit dữ liệu).
  - Thanh ghi dịch dữ liệu RX/sd\_reg hoặc RX/sd\_next thay đổi liên tục theo từng bit nhận, đồng thời RX/data\_out cũng thể hiện các giá trị trung gian trong quá trình “ghép byte”. Ngay trong hình có thể thấy chuỗi giá trị ví dụ như:
    - RX/data\_out: 00011100 → 00001110 → 00000111 → 00000011 → 00000001 → 10000000 → 01000000 → 10100000 → 01010000 ...
  - Các giá trị này xuất hiện đúng với hành vi shift/assemble dữ liệu trong lúc FSM đang ở state data.
- Chuyển sang PARITY (state = parity)
  - Sau khi RX/n\_reg chạy đến 0111 (đủ bit dữ liệu), RX/state\_reg chuyển sang parity (trong hình có chữ “parity”).
  - Tại thời điểm đang đứng ở vùng parity (theo cột Value bên trái):
    - RX/state\_reg = parity, RX/state\_next = parity
    - RX/data\_out = 11000000 (0xC0)
    - RX/sd\_reg = 11000000
    - RX/parity\_reg = 1, RX/parity\_next = 1
    - RX/incorrect\_send = 0 → không báo lỗi parity/frame ở đoạn này.
- Chuyển sang STOP rồi START cho frame kế tiếp
  - Ngay sau parity, waveform thể hiện tiếp stop, rồi start, rồi quay về data (đúng chuỗi chữ trên RX/state\_reg/state\_next: parity → stop → start → data).
  - Đây là đúng quy trình RX UART: nhận đủ data → kiểm tra parity → kiểm tra stop bit → bắt đầu frame mới.

### Hành vi FIFO\_RX (theo các tín hiệu FIFO trong hình)

- FIFO đang có dữ liệu (không rỗng)

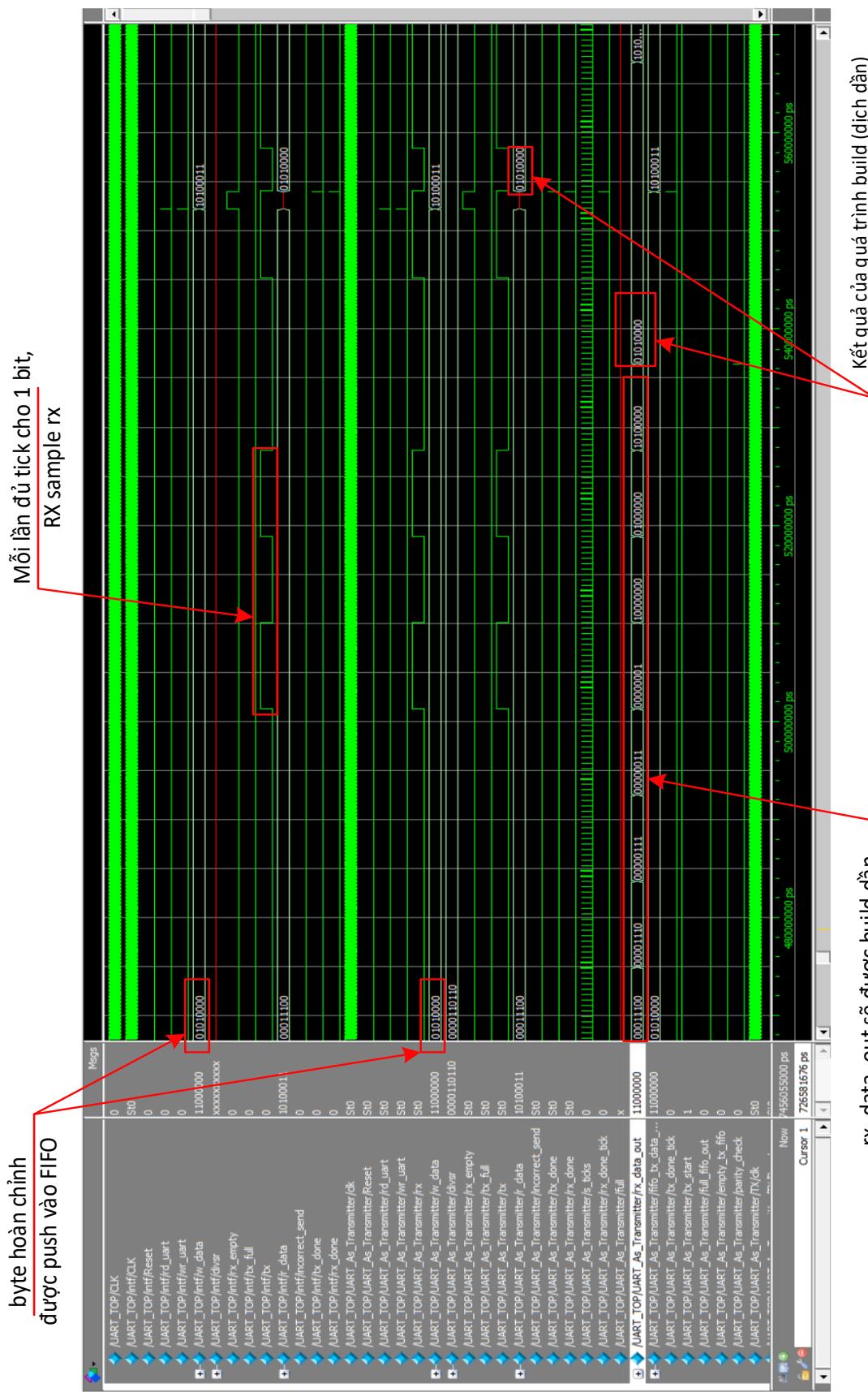
- Ở thời điểm con trỏ, cột Value cho thấy:
  - FIFO\_RX/empty = 0 → FIFO không rỗng
  - FIFO\_RX/full = 0 → FIFO chưa đầy
  - FIFO\_RX/w\_addr = 00110, FIFO\_RX/r\_addr = 00101
- Việc w\_addr lớn hơn r\_addr đúng với việc FIFO đã được ghi trước đó và vẫn còn dữ liệu chưa đọc.
- Đọc FIFO (pop)
  - Ở vùng bên phải hình, có một xung rõ ràng trên FIFO\_RX/rd (đọc FIFO).
  - Ngay tại thời điểm rd được kích:
    - FIFO\_RX/r\_data cập nhật (trong hình thấy nó nhảy về giá trị z01010000 = 01010000 (0x50))
    - FIFO\_RX/r\_addr tăng (trong hình thể hiện bước 00100 → 00101)
  - Điều này đúng với cơ chế FIFO: có rd → xuất r\_data → tăng địa chỉ đọc

### Tùy Hình 4.2 có thể kết luận:

- FSM RX chuyển trạng thái đúng theo tiến trình khung UART (Start → Data → Parity → End).
- w\_data phản ánh đúng quá trình thu từng bit và dịch trong RX.
- Khi RX hoàn tất, dữ liệu được đẩy vào FIFO\_RX, rx\_empty đổi trạng thái phù hợp.
- Khi có lệnh đọc, FIFO\_RX/r\_data xuất dữ liệu đúng cơ chế extract/pop.

### 4.3. DÒNG DỮ LIỆU TRONG THANH GHI DỊCH RX (rx\_data\_out)

Quan sát waveform trong **Hình 4.3** (đang zoom vào nhóm tín hiệu của UART\_As\_Transmitter) cho thấy rx\_data\_out không phải “dữ liệu cuối” ngay lập tức, mà nó hoạt động đúng kiểu shift register: mỗi lần RX lấy mẫu 1 bit (theo tick baud), thanh ghi sẽ dịch 1 bit và nhét bit mới vào.



Hình 4.3 – Kết quả mô phỏng “Data Flow”

## CHƯƠNG 4: KẾT QUẢ VÀ ĐÁNH GIÁ

---

**Hình 4.3** cho thấy tín hiệu *rx\_data\_out* thể hiện quá trình dịch bit để ghép byte dữ liệu ở phía RX. *rx\_data\_out* dịch bit theo từng lần lấy mẫu.

- Trên bus UART\_As\_Transmitter/*rx\_data\_out* thấy chuỗi giá trị thay đổi tuần tự:
  - 00001100 → 00000110 → 00000011 → 00000001 → 10000000 → 01000000  
→ 10100000 → 01010000 → ...
- Chuỗi này thể hiện rất rõ đúng cơ chế dịch phải 1 bit (ví dụ  $00001100 \gg 1 = 00000110$ ,  $00000110 \gg 1 = 00000011$ , ...) và tại một số bước MSB đổi (như ... → 10000000) nghĩa là bit mới nhận được đã được chèn vào đầu thanh ghi.

=> Đây chính là “data flow”: dữ liệu chảy qua thanh ghi dịch cho tới khi ghép đủ 8 bit.

- Khi ghép đủ byte, *rx\_data\_out* ổn định thành 1 giá trị hoàn chỉnh.

Ở cuối chuỗi trên hình, *rx\_data\_out* giữ được giá trị 01010000 (0x50) trong một đoạn, cho thấy RX đã assemble xong một byte (tức là sau khi nhận đủ bit data).

### 4.4. KẾT QUẢ BÁO CÁO QUA QUAN SÁT LOG MÔI TRƯỜNG

Quan sát log trong **Hình 4.4** cho thấy môi trường UVM in ra liên tục các bản tin UVM\_INFO từ scoreboard trong suốt quá trình mô phỏng. Các dòng log có dạng:

- Nguồn log: *uvm\_test\_top.UART\_env.UART\_scoreboard [scoreboard]*
- Nội dung chính: MATCH x1 value=0x.. kèm theo mốc thời gian và biến đếm *match\_total*.

Trong quá trình chạy, *match\_total* tăng dần theo từng byte được so sánh thành công (trong hình thấy các mốc như *match\_total*=1, ..., *match\_total*=2444). Điều này thể hiện cơ chế scoreboard hoạt động đúng: mỗi khi byte nhận được (RX) khớp với byte mong đợi (expected/TX) thì scoreboard báo MATCH và tăng bộ đếm.

Đặc biệt, ở phần cuối log (ngay trong **Hình 4.4**) có dòng tổng kết của scoreboard:

- SCOREBOARD SUMMARY Total Matches = 2444 Miss Matches = 0

=> Kết quả này khẳng định toàn bộ dữ liệu kiểm tra trong phiên mô phỏng đều khớp 100%. Ngoài ra, log cũng hiển thị trạng thái kết thúc test qua ID [TEST\_DONE] với thông báo kiểu “run phase is ready to proceed to the extract phase”, cho thấy test đã chạy xong và chuyển sang các phase kết thúc theo luồng UVM.

## CHƯƠNG 4: KẾT QUẢ VÀ ĐÁNH GIÁ

Hình 4.4. Kết quả báo cáo chạy mô phỏng qua màn hình LOG trên QuestaSim

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 2317
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
# ** Report counts by id
# [Questa UVM] 2
# [RNTST] 1
# [TEST_DONE] 1
# [UVMTOP] 1
# [scoreboard] 2309
# [sequencer] 3
# ** Note: $finish : C:/questasim64_2024.1/win64/..verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#   Time: 225825725 ns Iteration: 63 Instance: /UART_TOP
```

Hình 4.5. Kết quả UVM report summary

Sau khi mô phỏng kết thúc, **UVM Report Summary** tổng hợp số lượng report theo mức độ severity như sau (đúng theo hình):

- **UVM\_INFO : 2317**
- **UVM\_WARNING : 0**
- **UVM\_ERROR : 0**
- **UVM\_FATAL : 0**

Đồng thời phần “Report counts by id” cho thấy các bản tin chủ yếu đến từ:

- [scoreboard] : 2309 (chiếm đa số do in log MATCH liên tục)
- [sequencer] : 3
- [TEST\_DONE] : 1
- [RST] : 1
- và một số ID hệ thống khác (ví dụ Questa ..., UVM/TOPO).

=> Như vậy, báo cáo tổng kết xác nhận mô phỏng chạy ổn định, không phát sinh **warning/error/fatal**, scoreboard cho kết quả **2444 match – 0 mismatch**, chứng minh môi trường verify UART đang hoạt động đúng và testcase pass theo kỳ vọng.

## CHƯƠNG 4: KẾT QUẢ VÀ ĐÁNH GIÁ

### 4.5. KẾT QUẢ BÁO CÁO COVERAGE

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
UART_pkg/coverage_collector		100.00%				
TYPE UART_cg		100.00%	100	100.00%		✓
CVP UART_cg::KIND		100.00%	100	100.00%		✓
CVP UART_cg::RESET		100.00%	100	100.00%		✓
CVP UART_cg::W_DATA		100.00%	100	100.00%		✓
CVP UART_cg::R_DATA		100.00%	100	100.00%		✓
CVP UART_cg::TX_FULL		100.00%	100	100.00%		✓
CVP UART_cg::WDATA_STATUS		100.00%	100	100.00%		✓
CVP UART_cg::RX_EMPTY		100.00%	100	100.00%		✓
CVP UART_cg::INCORRECT_SEND		100.00%	100	100.00%		✓
CVP UART_cg::TX_DONE		100.00%	100	100.00%		✓
CVP UART_cg::RX_DONE		100.00%	100	100.00%		✓
CVP UART_cg::DIVSR		100.00%	100	100.00%		✓
CROSS_UART_cg::CROSS_WDATA_FULL		100.00%	100	100.00%		✓
INST \UART_pkg::coverage_collector::UART_cg		100.00%	100	100.00%		✓

Hình 4.6. Tóm tắt kết quả báo cáo coverage cho verify plan đã lên

Kết quả báo cáo coverage của môi trường verify UART được thể hiện trong **Hình 4.6**. Từ bảng thống kê cho thấy covergroup **UART\_cg** đạt 100%, và tất cả các coverpoint đều đã được hit đầy đủ, bao gồm: *KIND*, *RESET*, *W\_DATA*, *R\_DATA*, *TX\_FULL*, *WDATA\_STATUS*, *RX\_EMPTY*, *INCORRECT\_SEND*, *TX\_DONE*, *RX\_DONE*, *DIVSR*. Bên cạnh đó, cross coverage *CROSS\_WDATA\_FULL* cũng đạt 100%. Như vậy, coverage tổng thể của verify plan đạt 100%, chứng tỏ các trường hợp kiểm thử đã bao phủ đầy đủ các mục tiêu đặt ra.

Bảng 4.1. Kết quả báo cáo coverage cho covergroup **UART\_cg**

COVERGROUP COVERAGE:				
Covergroup	Metric	Goal	Bins	Status
TYPE /UART_pkg/coverage_collector/UART_cg	100.00%	100	-	Covered
covered/total bins:	40	40	-	
missing/total bins:	0	40	-	
% Hit:	100.00%	100	-	
Coverpoint KIND	100.00%	100	-	Covered
covered/total bins:	4	4	-	
missing/total bins:	0	4	-	
% Hit:	100.00%	100	-	
bin rst	3	1	-	Covered
bin wr	7332	1	-	Covered
bin rd	7332	1	-	Covered
bin status	29820	1	-	Covered
Coverpoint RESET	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin deasserted	44484	1	-	Covered
bin asserted	3	1	-	Covered
Coverpoint W_DATA	100.00%	100	-	Covered
covered/total bins:	4	4	-	
missing/total bins:	0	4	-	
% Hit:	100.00%	100	-	
bin bin_all_zeros	120	1	-	Covered
bin bin_all_ones	123	1	-	Covered
bin bin_pat_aa	129	1	-	Covered
bin bin_pat_55	114	1	-	Covered
default bin default_bin_others	6846	-	-	Occurred

## CHƯƠNG 4: KẾT QUẢ VÀ ĐÁNH GIÁ

---

Coverpoint R_DATA	100.00%	100	-	Covered
covered/total bins:	4	4	-	
missing/total bins:	0	4	-	
% Hit:	100.00%	100	-	
bin bin_all_zeros	120	1	-	Covered
bin bin_all_ones	123	1	-	Covered
bin bin_pat_aa	129	1	-	Covered
bin bin_pat_55	114	1	-	Covered
default bin default_bin_others	6846	-	-	Occurred
Coverpoint TX_FULL	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin bin_not_full	29772	1	-	Covered
bin bin_full	48	1	-	Covered
Coverpoint WDATA_STATUS	100.00%	100	-	Covered
covered/total bins:	4	4	-	
missing/total bins:	0	4	-	
% Hit:	100.00%	100	-	
bin bin_all_zeros	498	1	-	Covered
bin bin_all_ones	690	1	-	Covered
bin bin_pat_aa	684	1	-	Covered
bin bin_pat_55	690	1	-	Covered
default bin default_bin_others	27258	-	-	Occurred
Coverpoint RX_EMPTY	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin bin_not_empty	7332	1	-	Covered
bin bin_empty	22488	1	-	Covered
Coverpoint INCORRECT_SEND	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin bin_ok	25914	1	-	Covered
bin bin_err	3906	1	-	Covered
Coverpoint TX_DONE	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin low	22488	1	-	Covered
bin high	7332	1	-	Covered
Coverpoint RX_DONE	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin low	22488	1	-	Covered
bin high	7332	1	-	Covered
Coverpoint DIVSR	100.00%	100	-	Covered
covered/total bins:	4	4	-	
missing/total bins:	0	4	-	
% Hit:	100.00%	100	-	
bin d54	9940	1	-	Covered
bin bin_small	9940	1	-	Covered
bin bin_mid	9940	1	-	Covered
bin bin_big	9940	1	-	Covered
Cross CROSS_WDATA_FULL	100.00%	100	-	Covered
covered/total bins:	8	8	-	
missing/total bins:	0	8	-	
% Hit:	100.00%	100	-	
Auto, Default and User Defined Bins:				
bin all_zeros_not_full	486	1	-	Covered
bin all_zeros_full	12	1	-	Covered
bin all_ones_not_full	678	1	-	Covered
bin all_ones_full	12	1	-	Covered
bin pat_aa_not_full	672	1	-	Covered
bin pat_aa_full	12	1	-	Covered
bin pat_55_not_full	678	1	-	Covered
bin pat_55_full	12	1	-	Covered
Covergroup instance \UART_pkg::coverage_collector::UART_cg	100.00%	100	-	Covered
covered/total bins:	40	40	-	
missing/total bins:	0	40	-	
% Hit:	100.00%	100	-	
Coverpoint KIND	100.00%	100	-	Covered
covered/total bins:	4	4	-	
missing/total bins:	0	4	-	
% Hit:	100.00%	100	-	
bin rst	3	1	-	Covered
bin wr	7332	1	-	Covered
bin rd	7332	1	-	Covered
bin status	29820	1	-	Covered
Coverpoint RESET	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin deasserted	44484	1	-	Covered
bin asserted	3	1	-	Covered

## CHƯƠNG 4: KẾT QUẢ VÀ ĐÁNH GIÁ

---

Coverpoint W_DATA	100.00%	100	-	Covered
covered/total bins:	4	4	-	
missing/total bins:	0	4	-	
% Hit:	100.00%	100	-	
bin bin_all_zeros	120	1	-	Covered
bin bin_all_ones	123	1	-	Covered
bin bin_pat_aa	129	1	-	Covered
bin bin_pat_55	114	1	-	Covered
default bin default_bin_others	6846		-	Occurred
Coverpoint R_DATA	100.00%	100	-	Covered
covered/total bins:	4	4	-	
missing/total bins:	0	4	-	
% Hit:	100.00%	100	-	
bin bin_all_zeros	120	1	-	Covered
bin bin_all_ones	123	1	-	Covered
bin bin_pat_aa	129	1	-	Covered
bin bin_pat_55	114	1	-	Covered
default bin default_bin_others	6846		-	Occurred
Coverpoint TX_FULL	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin bin_not_full	29772	1	-	Covered
bin bin_full	48	1	-	Covered
Coverpoint WDATA_STATUS	100.00%	100	-	Covered
covered/total bins:	4	4	-	
missing/total bins:	0	4	-	
% Hit:	100.00%	100	-	
bin bin_all_zeros	498	1	-	Covered
bin bin_all_ones	690	1	-	Covered
bin bin_pat_aa	684	1	-	Covered
bin bin_pat_55	690	1	-	Covered
default bin default_bin_others	27258		-	Occurred
Coverpoint RX_EMPTY	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin bin_not_empty	7332	1	-	Covered
bin bin_empty	22488	1	-	Covered
Coverpoint INCORRECT_SEND	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin bin_ok	25914	1	-	Covered
bin bin_err	3906	1	-	Covered
Coverpoint TX_DONE	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin low	22488	1	-	Covered
bin high	7332	1	-	Covered
Coverpoint RX_DONE	100.00%	100	-	Covered
covered/total bins:	2	2	-	
missing/total bins:	0	2	-	
% Hit:	100.00%	100	-	
bin low	22488	1	-	Covered
bin high	7332	1	-	Covered
Coverpoint DIVSR	100.00%	100	-	Covered
covered/total bins:	4	4	-	
missing/total bins:	0	4	-	
% Hit:	100.00%	100	-	
bin d54	9940	1	-	Covered
bin bin_small	9940	1	-	Covered
bin bin_mid	9940	1	-	Covered
bin bin_big	9940	1	-	Covered
Cross CROSS_WDATA_FULL	100.00%	100	-	Covered
covered/total bins:	8	8	-	
missing/total bins:	0	8	-	
% Hit:	100.00%	100	-	
Auto, Default and User Defined Bins:				
bin all_zeros_not_full	486	1	-	Covered
bin all_zeros_full	12	1	-	Covered
bin all_ones_not_full	678	1	-	Covered
bin all_ones_full	12	1	-	Covered
bin pat_aa_not_full	672	1	-	Covered
bin pat_aa_full	12	1	-	Covered
bin pat_55_not_full	678	1	-	Covered
bin pat_55_full	12	1	-	Covered

TOTAL COVERGROUP COVERAGE: 100.00% COVERGROUP TYPES: 1

## CHƯƠNG 4: KẾT QUẢ VÀ ĐÁNH GIÁ

---

Kết quả báo cáo coverage cho covergroup **UART\_cg** được thể hiện như **bảng 4.1**: tổng coverage đạt 100%, với 40/40 bins đã được hit và 0 bins bị thiếu. Điều này chứng tỏ các mục tiêu trong **verify plan** đã được kích hoạt đầy đủ trong quá trình mô phỏng.

Cụ thể, các coverpoint đều đạt 100%:

- KIND gồm 4 bins (*rst*, *wr*, *rd*, *status*) đều được thực thi; trong đó các thao tác *wr* và *rd* xuất hiện nhiều lần (7332 lần), và thao tác *status* xuất hiện 29820 lần.
- RESET bao phủ đủ 2 trạng thái *asserted* và *deasserted*, cho thấy test đã có pha reset và pha hoạt động bình thường.
- W\_DATA và R\_DATA đều hit đủ các pattern mục tiêu (*all\_zeros*, *all\_ones*, *0xAA*, *0x55*), đồng thời các giá trị còn lại cũng xuất hiện qua *default\_bin\_others*.
- Các trạng thái FIFO/handshake như TX\_FULL, RX\_EMPTY, TX\_DONE, RX\_DONE đều bao phủ đủ cả 2 mức (low/high hoặc empty/not\_empty, full/not\_full).
- INCORRECT\_SEND hit được cả trường hợp OK và ERR, cho thấy môi trường đã quan sát được cả tình huống truyền đúng và tình huống lỗi theo tiêu chí coverage.
- DIVSR bao phủ đủ 4 bins (d54, small, mid, big), chứng tỏ test đã chạy ở nhiều cấu hình baud divisor.

Ngoài ra, cross coverage CROSS\_WDATA\_FULL đạt 100% với 8/8 bins, bao phủ đầy đủ các kết hợp giữa pattern dữ liệu (00/FF/AA/55) và trạng thái TX\_FULL (full/not\_full).

=> **Tổng kết lại**, TOTAL COVERGROUP COVERAGE = 100% với 1 covergroup type, xác nhận môi trường verify UART đã đạt mức bao phủ đầy đủ theo kế hoạch kiểm thử.

## CHƯƠNG 5: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

Source Code đồ án tại: [https://github.com/QuangHop-dev/UART\\_Verification\\_using\\_UVM](https://github.com/QuangHop-dev/UART_Verification_using_UVM)

### 5.1. KẾT LUẬN

Đề tài “**Thiết kế và xác minh chức năng UART bằng UVM**” đã xây dựng được một môi trường kiểm chứng theo phương pháp UVM cho IP UART. Môi trường được tổ chức đầy đủ các thành phần chính gồm **sequence/sequence\_item**, **sequencer**, **driver**, **monitor**, **scoreboard** và **coverage collector**, cho phép tạo stimulus, thu thập tín hiệu quan sát, so sánh expected–actual và đo lường mức độ bao phủ theo verify plan.

- Kết quả mô phỏng cho thấy hệ thống kiểm chứng hoạt động ổn định.
- Functional coverage đạt 100% với 40/40 bins của covergroup UART\_cg và 8/8 bins của cross CROSS\_WDATA\_FULL.
- Báo cáo UVM tổng kết không phát sinh warning/error/fatal, cho thấy mô phỏng chạy xuyên suốt và môi trường testbench không bị lỗi vận hành.

#### Ưu điểm:

- Mô hình UVM được xây dựng đúng cấu trúc chuẩn (agent-env-test), dễ mở rộng/regression.
- Driver hỗ trợ nhiều “chế độ chạy” qua plusargs (timeout, passive/active, random/direct, inject RX...), thuận tiện cho đóng coverage.
- Monitor đã tách rõ WR/RD transaction và STATUS transaction, giúp coverage bám sát verify plan và scoreboard so sánh đúng thời điểm.

#### Hạn chế / Chưa đạt được:

- Coverage 100% mới phản ánh đúng các bins đã định nghĩa, chưa đảm bảo đã quét hết corner-case theo nghĩa “đặc tả đầy đủ” (ví dụ: nhiều mẫu dữ liệu hơn, timing/divsr đa dạng hơn, trường hợp nhiều start/stop, back-to-back frame dài...).
- Chưa tích hợp assertion (SVA) và chưa tổng hợp thêm code coverage (line/branch/toggle) để đánh giá toàn diện hơn.
- Môi trường hiện tại chỉ kiểm tra được các hành vi của UART trong chế độ loopback.

- Do giới hạn thời gian và phạm vi đề tài, môi trường hiện chưa xây dựng mô hình kiểm thử cho hai UART độc lập (UART A truyền sang UART B qua đường serial như kịch bản hệ thống thực tế). Vì vậy, các trường hợp liên quan đến sai lệch baud giữa hai phía, jitter/timing, bát đồng bộ clock, hoặc kiểm tra tương tác giữa hai instance UART chưa được đánh giá đầy đủ.

## 5.2. HƯỚNG PHÁT TRIỂN

Trong tương lai, đề tài có thể phát triển theo các hướng sau để tăng độ tin cậy và tiệm cận quy trình DV thực tế:

- Xây dựng **môi trường kiểm thử 2 UART** (UART-to-UART): instantiate hai DUT UART độc lập (UART\_A và UART\_B), nối tx\_A → rx\_B và tx\_B → rx\_A, từ đó kiểm tra truyền thông hai chiều theo kịch bản thực tế thay vì loopback.
- Mở rộng verify plan & testcase corner-case
- Nâng cấp scoreboard theo hướng “reference model”
- Assertion-Based Verification (SVA)
- Automation regression & closure: Viết script chạy nhiều test/seed, tự động save + merge UCDB, xuất report coverage/log summary để phục vụ closure nhanh và có thể tái lập kết quả.
- Mở rộng mô hình giao tiếp mức hệ thống: Nếu DUT có register/bus (APB/Wishbone/AXI-lite), có thể bổ sung BFM + UVM RAL để verify đường cầu hình (divsr, enable, interrupt...) theo đúng flow SoC.

## TÀI LIỆU THAM KHẢO

- [1] R. H. H. S. Prasad, C. Santhi Rani, “*UART IP Core Verification by Using UVM*,” in **Proceedings of 42nd IRF International Conference**, Chennai, India, 15th May 2016, p. 30, ISBN: 978-93-86083-17-3.
- [2] N. B. S. Niranjan, S. K. Panda, S. V. A. Shanthi, “*Design and Development of Verification Environment to Verify UART Protocol using UVM*,” **International Journal of VLSI System Design and Communication Systems**, vol. 03, issue 03, pp. 0262–0264, June 2015, ISSN: 2322-0929.
- [3] K. K. Chavan, S. S. Kirdak, A. S. Bhagwat, S. R. Sabale, A. Gangad, “*Design and Development of UART Protocol Using Verilog with UVM Testbench*,” **International Journal of Scientific Research in Engineering and Management (IJSREM)**, vol. 08, issue 05, May 2024, ISSN: 2582-3930.
- [4] M. Srinath, S. Sujatha Hiremath, “*Verification of Universal Asynchronous Receiver and Transmitter (UART) using System Verilog*,” **International Journal of Engineering Research & Technology (IJERT)**, vol. 11, issue 07, July 2022, ISSN: 2278-0181.
- [5] R. Salemi, *The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology*, 1st ed. Boston Light Press, 2013.
- [6] M. Horn, M. Peryer, T. Fitzpatrick and J. Stickley, *Universal Verification Methodology UVM Cookbook*.
- [7] Siemens EDA, “*QuestaSim Tutorial – Including Support for Questa Base*,” **Siemens Digital Industries Software**, Nov. 2024. [Online]. Available: <https://eda.sw.siemens.com>
- [8] Siemens EDA, “*QuestaSim Command Reference Manual – Including Support for Questa Base*,” **Siemens Digital Industries Software**, Nov. 2024. [Online]. Available: <https://eda.sw.siemens.com>

# PHU LUC

Test Case ID	Category	Stimulus	RX action	RX action	Date/Pattern	Error inject	Expected / Check	Feature
2 TC01	RESET	Apply reset (Reset=1 for 5 cycles then 0)	54 -	-	-	-	rx_empty=1; tx_full=0; tx=1	Reset defaults
TC02	BAUD	Set divisor=54 then run a short TX+RX activity	54 wr_uart + inject_rx_frame	rd_uart 0x55	-	-	No timeout; s_ticks timing changes with divisor	Baud tick generation
3 TC03	BAUD	Set divisor=2 then run a short TX+RX activity	2 wr_uart + inject_rx_frame	rd_uart 0x55	-	-	No timeout; s_ticks timing changes with divisor	Baud tick generation
4 TC04	BAUD	Set divisor=10 then run a short TX+RX activity	10 wr_uart + inject_rx_frame	rd_uart 0x55	-	-	No timeout; s_ticks timing changes with divisor	Baud tick generation
5 TC05	BAUD	Set divisor=200 then run a short TX+RX activity	200 wr_uart + inject_rx_frame	rd_uart 0x55	-	-	No timeout; s_ticks timing changes with divisor	Baud tick generation
6 TC06	TX_BASIC	Write byte 0x00 to FIFO_TX and observe tx waveform	54 wr_uart pulse	-	0x00	-	TX frame = start[0]+8 data LSB-first+parity+stop[1]; tx_done_tick asserts at end	TX serialization/bit order
7 TC07	TX_BASIC	Write byte 0xFF to FIFO_RX and observe tx waveform	54 wr_uart pulse	-	0xFF	-	TX frame = start[0]+8 data LSB-first+parity+stop[1]; tx_done_tick asserts at end	TX serialization/bit order
8 TC08	TX_BASIC	Write byte 0xAA to FIFO_RX and observe tx waveform	54 wr_uart pulse	-	0xAA	-	TX frame = start[0]+8 data LSB-first+parity+stop[1]; tx_done_tick asserts at end	TX serialization/bit order
9 TC09	TX_BASIC	Write byte 0x55 to FIFO_RX and observe tx waveform	54 wr_uart pulse	-	0x55	-	TX frame = start[0]+8 data LSB-first+parity+stop[1]; tx_done_tick asserts at end	TX serialization/bit order
10 TC10	FIFO_RX	Fill FIFO_RX until tx_full==1	54 Burst writes	-	random bytes	-	tx_full asserts; additional writes are blocked (w_en gated)	FIFO full flag
11 TC11	FIFO_RX	After full, allow UART_RX to drain and check tx_full	54 Wait tx_done / allow drain	-	-	-	tx_full eventually returns 0 as FIFO drained	FIFO drain behavior
12 TC12	RX_BASIC	Inject RX frame 0x00 (good parity) then pop FIFO_RX	54 -	inject_rx_frame + rd_uart	0x00	-	r_data==injected byte; rx_done observed; rx_empty toggles 1>0->1	RX capture + FIFO_RX
13 TC13	RX_BASIC	Inject RX frame 0xFF (good parity) then pop FIFO_RX	54 -	inject_rx_frame + rd_uart	0xFF	-	r_data==injected byte; rx_done observed; rx_empty toggles 1>0->1	RX capture + FIFO_RX
14 TC14	RX_BASIC	Inject RX frame 0xAA (good parity) then pop FIFO_RX	54 -	inject_rx_frame + rd_uart	0xAA	-	r_data==injected byte; rx_done observed; rx_empty toggles 1>0->1	RX capture + FIFO_RX
15 TC15	RX_BASIC	Inject RX frames 0x55 (good parity) then pop FIFO_RX	54 -	inject_rx_frame + rd_uart	0x55	-	r_data==injected byte; rx_done observed; rx_empty toggles 1>0->1	RX capture + FIFO_RX
16 TC16	RX_ERROR	Inject RX frame with bad parity then pop	54 -	inject_rx_frame[ba d_parity!=1]+rd_uart	0xA5	bad_parity	incorrect_send asserted (per design); data still captured	Parity error flag
17 TC17	RX_ERROR	Inject RX frame with bad stop bit (stop=0)	54 -	inject_rx_frame[ba d_stop!=1]+rd_uart	0x3C	bad_stop	rx_done still asserts (no stop checking in RTL); incorrect_send depends only on Parity	Stop-bit handling
18 TC18	NEG	Assert rd_uart while rx_empty==1	54 -	rd_uart pulse (no data)	-	-	No valid rx_empty remains 1; rx_data not considered valid	Graceful empty read
19 TC19	NEG	Attempt write when tx_full==1 (requires special sequence bypass wait)	54 wr_uart while tx_full==1	-	0x5A	-	Write should be blocked by w_en=wr&~tx_full; FIFO content unchanged	Graceful full write
20 TC20	STRESS	Random mix: write/read/inject with random delays	54 random writes	random reads/inject	optional bad_parity	0x00..0x07	Scoreboard has zero mismatches; no timeouts; coverage increases	Constrained-random Stability
21 TC21	RX_STRESS	Inject 8 back-to-back RX frames then read all	54 -	inject_rx_frame x8 + rd_uart	-	until drained	FIFO_RX stores all bytes in order; rx_empty deasserts	RX throughput
22 TC22	TX_STRESS	Queue 8 TX writes back-to-back then observe continuous tx	54 wr_uart x8	-	0x10..0x17	-	TX sends frames sequentially; tx_done pulses per frame; tx_full behaves	TX throughput
23 TC23	FLAGS	Check tx_done_tick is 1-cycle pulse	54 single write	-	0xC3	-	tx_done_tick pulses for 1 clk (or 1 sample) at end of stop	Done pulse width
24 TC24	FLAGS	Check rx_done_tick is 1-cycle pulse	54 -	inject good frame	0xC3	-	rx_done_tick pulses at end of stop	Done pulse width
25 TC25	COV	Generate bins for VDATA_STATUS x TX_FULL cross	54 writes across patterns & FIFO fill	0x00/F/AA/55	-	-	Hit all bins in coverage_collector	Functional coverage cross

**Fig. UART Verification Plan**